Operating System Specification Using Very High Level Dictions

by

Peter Markstein
June, 1975

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Sciences in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University.

Approved by: _____
Prof. Jacob T. Schwartz

Operating System Specification Using Very High Level Dictions

by
Peter Markstein

ABSTRACT

Today, operating systems are an integral part of computing systems. Yet high level programming languages are not generally used by the designers and implementors of systems software. In this report we intoduce a language, PSETL, intended for operating system description and based on the SETL set-theoretic programming language.

PSETL is a version of SETL which has been enlarged to allow the description of algorithms involving interrupts, parallelism, and to some extent, machine dependent features. Using PSETL, several operating systems are presented in detail. The first, a simple uniprogrammed batch system illustrates basic control mechanisms and scheduling. The second, a multiprogrammed batch system, shows additional complications which arise due to contention for resources and conflicting objectives. Our third system is interactive and includes data sharing capabilities.

Research Advisor:
Prof. Jacob T. Schwartz

PREFACE

This thesis is an experiment in extending a very high level language, SETL, for operating system description. We propose an extension to SETL, and then use our extension to specify several operating systems. The principal question we are trying to address is: can extended SETL be used for the specification of operating systems as effectively as SETL can be used for other classes of problems? The reader, after studying the examples of Chapters III, IV, and V will have formed his answer to this question.

An experiment of this kind cannot confine itself to the discussion of small examples. Operating systems are inherently larger and more complex than, say, sorting algorithms. To give a fair demonstration of extended SETL we feel it necessary to describe an entire operating system; even a simple example of this kind is relatively large, and requires at least a dozen pages of code.

If our experiment is successful, then detailed specifications of several operating systems, in comprehensible form, appear in this work. In order to motivate the examples which are presented, and make them as clear as possible, this thesis has been cast in the form of an introductory text on operating systems which contains three completely coded examples. Of course, it is these examples which make the text unique.

The reader is assumed to have a working knowledge of SETL; no explanations of SETL coding dictions appear in the body of the text. On the other hand, no previous exposure to operating system internals is assumed on the part of the reader.

Acknowledgements

I am indebted to Professor Schwartz for his many useful suggestions during the preparation of this work.

My employer, International Business Machines, Inc. has very generously given support to this work through its Graduate Work-Study Program.

Special thanks go to my wife, Vicky, for her encouragement and patience during my graduate years. And apologies are due to my daughters Carole and Michele for my occasional preoccupation with my studies.

CONTENTS

Chapter I

Introduction to Operating Systems

Our first task is to define what an operating system is. It will not be possible to do this with mathematical precision. Instead, a loose characterization of 'operating system' will be given, motivated by citing objectives which such systems attempt to satisfy.

1.1 Informal Review of Operating System Objectives

1.1.1 The Automatic Operator

In the early days of computing each job or run was an independent entity. A user submitted his own copy of a language processor, loader, or debugging aid, along with instructions for the operator on actions to take on the occurrence of various machine halts. At the end of a run, computer memory was generally cleared, tape reels associated with the concluded run dismounted, and tape reels for the next run mounted. Transition time between runs was frequently of the order of 1 to 5 minutes. These inefficiencies were often compounded by the inability of a computing installation to process a multi-step job. Thus, a "compile and go" job was usually two runs, with the attendant overhead paid twice.

With a larger number of applications becoming economically feasible and with increased computer speed, the length of typical computer runs - especially runs for debugging - approached and fell below the run transition time. Just as the human's ability to enter data and commands became the limiting factor for desk calculators, so the speed of humans during run transition time threatened to become the limiting factor in the use of computers.

As with conventional computing, the solution to the job transition problem used the stored program concept. Information describing characteristics of a job and the relations between job steps are included in machine readable form along with the data and programs which comprise the job. A computer program given a sequence of jobs which include job characteristic information can then determine an efficient order in which to run jobs. Such a program is commonly called an operating system. (The additonal statements specifying job characteristics and other operational information constitute the 'job control language'.) Multiple job steps per run become more common as a machine, instead of an operator, interprets and acts upon conditions stating whether subsequent steps should be executed, and transmits the output of one step to the next. By using precisely stated job or step transition information and resource requirements a computer program can take many of the actions previously associated with human operators, reducing job transition time to a few seconds at most.

Success of the programmed or automatic operator depends on control returning to the operating system at the conclusion of a job or step. This is insured either through software conventions, special hardware, or both. Modern computers have hardware facilities whose use guarantees integrity of the operating system, and enforces its software conventions.

1.1.2 Program Libraries

In any computer installation, there are a number of programs which are useful to a large class of users. Examples of such programs are language processors, loaders, debugging aids, as well as

application programs such as sorting programs. Rather than require each user to supply his own copy of such programs, an installation maintains a library of these frequently used programs, and an operating system can invoke these library programs on behalf of a user in response to job control language statements. Thus, instead of submitting a bulky program, a few JCL statements are all that a user need submit to invoke a library routine. Use of a centralized program library also insures that the most current version of a utility program is available to all.

Operating systems usually include facilities for updating and maintaining program libraries.

An obvious extension to the program library idea is to permit subgroups of users to create and maintain private libraries of programs. The same operating system facilities which are used to create and maintain the central library are usually available for the private libraries, and JCL generally invokes programs from any library with equal ease.

One aspect of program library maintenance should be mentioned at this point because of its utility in a wide class of situations: this is the data management capability of operating systems. Data management involves construction and maintenance of catalogues which can be used to locate users' files, and structuring data files so that specified subsets can be easily extracted. In its most primitve form, data management merely aids in the coding of complex input-output instructions; advanced data management makes available convenient and powerful linguistic devices for characterizing and extracting subsets from data files.

1.1.3 Resource Utilization

The discussion of automatic operation in section 1.1.1 indicated the need for a program to manage the sequencing of jobs in order to avoid excessive system idle time between runs. This function of operating systems, while it is the function which historically motivated their construction, is just one aspect of the more general problem of maximizing utilization of the entire computing system.

Many of today's computing systems include more equipment than any single job in the installation can use. Such configurations are justified by the desire to offer a wide class of services. For example, a large accounting application might require many tape or disc drives but not much main memory, while even a moderate linear programming problem can use a large main memory to advantage.

Most jobs, however, do not tax any one component of hardware to the utmost. For such large computing systems, running only one job at a time can result in a substantial portion of the computing system's resources standing idle. To increase total system utilization, operating systems exploit the fact that equipment other than CPU and main memory can operate autonomously from the CPU for myriads of CPU instruction cycles. Several jobs are placed into main memory concurrently and control of the CPU given to one of them. When the currently running job reaches a state where it cannot utilize the CPU until the termination of an I/O operation, the CPU can be exploited by one of the other jobs in main memory. On the other hand, it would be undesirable if each application were to be written in such a manner to cooperate only with a specific set of other applications, for then the economies of concurrent running can only be realized when all members of a set of cooperating programs run together. Ideally, it should be

possible to write a program as if it were the only program being run, and still realize economies if it fits into main memory with another program which has a different pattern of I/O usage.

Many operating systems permit precisely this type of programming. Using the interrupt facilities of the CPU, the operating system can gain control when a user program is about to become idle, and give control of the CPU to another job. Similarly, when an awaited condition is satisfied, the operating system can regain control, and then return control to the task which was waiting for the condition to be satisfied. The sharing of hardware resources by independent jobs in the manner just described is called multiprogramming.

To effectively schedule the typical mix of jobs present in a multiprogramming environment, the operating system may require characterizations of the jobs being submitted for execution. Such information can be supplied via the job control language. Assuming that more than enough work is available a possible objective of an operating system in scheduling jobs and determining which jobs are to run concurrently is to minimize the rental paid for idle equipment.

In practice, however, the system must take account of other constraints such as job deadlines. Indeed, if we take a broader view of a computing system and include as "components" the people whose activities depend on the results of computation, then their idle time must also be taken into account. A direct consequence of such reasoning is the construction of interactive computing or time-sharing systems which on the surface appear to require extra hardware which does not contribute directly to throughput.

The users of shared systems must have guarantees that their programs and data will not be disturbed by co-resident programs. This problem has already been alluded to in the discussion of automatic operation in section 1.1.1; the same techniques which guarantee integrity of the operating system in a uniprogrammed environment can be extended to prevent physical interference among multiprogrammed jobs.

1.1.4 Hardware Control

To guarantee the integrity of programs sharing a computing system, one removes direct control of some hardware features from the system's users, and makes those features available only through simulation, during which potential misuse can be detected and prevented.

The hardware features whose control the operating system reserves for itself are precisely those features which are used to subdivide the computing system's resources. The instruction set of today's computers usually consists of two classes: privileged and non-privileged instructions. The privileged instruction set includes facilities for input-output, instructions for setting access boundaries in main memory, interrupt control instructions, and instructions which transfer the computer between problem state and supervisor state. In the supervisor state, all instructions are valid; in the problem state, privileged instructions are treated as illegal and cause interrupts. The enabling and disabling of the interrupt system requires privileged instructions.

Input/output instructions are classified as privileged to prevent one user from accessing a device which contains data belonging to another user. In many cases, a single physical device, such as a disc, contains data belonging to several users, and the operating system is required to establish correct 'logical' to 'physical' correspondences. To replace the privileged I/O instructions which

users are not allowed to invoke directly, the operating system provides I/O routines. A benefit to the user is that the operating system also provides additional facilities which provide automatic buffering and synchronization between I/O and computing.

By reserving control of physical I/O addresses to the operating system, the chance that arbitrary programs can be multiprogrammed is maximized; for if two programs depended on using the same physical I/O unit from a set of identical devices, these two programs could never run concurrently. Job control language provides mechanisms to establish correspondences between user invented file names and the devices on which the files are located. A compensation for the loss of direct control of I/O devices is that the I/O instructions and I/O error indications provided by the operating system tend to be device independent, so that frequently a program can utilize a wide variety of devices for a temporary file without any modification.

The interrupt system is also privileged, as it is the principal means of communicating exceptional conditions to the operating system, including attempted violations of security. Often a simulated interrupt system is made available to user programs, so that these programs can also handle exceptional conditions without explicitly executing tests which usually fail. Of course, the fact that the operating system handles many classes of interrupts explicitly means that programs are relieved of all necessity to concern themselves with interrupts.

To allow sharing of main memory, users must often state how much contiguous space they will require, but usually do not have the freedom to specify the actual addresses in memory where the space will be. This limitation imposes minimal user discomfort, since the use of relocating loaders has already preempted some control over memory allocation. Even this modest level of discomfort is avoided in computing systems which have "virtual memory" capabilities. Such systems can often be programmed so that each user in a multiprogrammed environment has the illusion of having all the original resources of the computer, including all memory locations, at his disposal.

1.1.5 Preparation and Maintenance of Software

Once one has realised what the objectives of operating systems are, one is faced with the problem of producing the operating system itself. With what programming approach shall the initial design, development, and debugging of the operating system be attempted? What features for self maintenance shall be introduced into the operating system: what instrumentation, software error detection capability, what ability to test the operating system under itself? Ability of the operating system to accomodate a wide variety of hardware configurations is also an important design issue.

A significant fraction of an operating system consists of library programs which behave as user programs and which are maintained in the same way as ordinary user programs. Language processors are examples of such operating system components. Standard library facilities can also be used to maintain and update the source programs comprising the operating system, and the language processors to compile these programs. Furthermore, ordinary programs can be used to structure compiled operating system programs into a new operating system. The only portions of the operating system which are notably difficult to debug in an operating system environment are those routines involved with hardware and resource allocation.

1.2 Overview of Operating System Internals

Simulation of human operations, achieving hardware control and optimizing hardware utilization imply characteristics not often found in ordinary application programs. A human operator at the console of a computer exercises direct control only sporadically, while he observes the system continuously, looking for for unusual occurences. An operating system, which among other things simulates a human computer operator, must be able to exhibit similar behavior, that is, the operating system must give up control of the CPU for a majority of the time so that user programs can run, and in initiating a user program it must place the CPU in such a state such that if any of a number of special situations arise, control returns immediately to the operating system. Such behavior can be achieved by simulating successive user program instructions and testing for the unusual conditions as part of the simulator's basic cycle, but this is very inefficient. Computers which are designed to run with operating systems contain an interrupt system which makes it possible for crucial changes of CPU control to take place efficiently. Leaving the computers in a state enabled for all interrupt conditions is equivalent to constantly monitoring for unusual conditions but taking overt action only when such conditions occur. Control and management of the interrupt system is fundamental to an operating system.

Even the simplest operating system creates a multiprogramming environment in the sense that the operating system consists of several relatively autonomous subprograms which run "concurrently" and which have the property of requiring only short bursts of CPU usage between which only monitoring of unusual events is required. Examples of functions treated in this way include: scheduling and dispatching jobs, controlling input/output devices, requesting and confirming the mounting of tape reels or disc packs, and avoiding user program time overruns. Such functions are then multiprogrammed with one or more user programs.

1.2.1 The State of a Computation

A computer running under an operating system is actually involved with several potentially active computations at the same time. One of these programs may be in control of the CPU; the state of the other computations must be stored in a form allowing them to be restarted.

The detailed description of a computation's state is machine dependent; however, it can be characterized With sufficient precision in an abstract way. The state of an interrupted computation consists of all the information necessary to resume the computation. At the machine level, this information falls into three broad categories: data resident in registers of the CPU, data resident in the address space of the computation, and data resident in files.

For each program which it manages, an operating system reserves a portion of memory, addressable only by the operating system, for storing the CPU resident data when the program loses control of the CPU. A typical item of register-resident data is the location at which to resume execution; this is the last datum which the operating system restores when returning control to a program.

In many batch systems, address space resident data remains physically resident in central memory during a program's entire run, including times when the program does not control the CPU. Alternatively, one can copy the memory resident data associated with a program P onto a file accessible only by the operating system when P loses control of the CPU, and can bring it back to physical central memory before returning control of the CPU to P. Other schemes involve

maintaining only a fraction of address space resident data in physical memory; this 'virtual memory' approach will be discussed in greater detail below.

Since physical devices may store files associated with independent programs, the operating system must keep track of the assumptions which each program makes about the logical positioning of such devices, so that these assumptions will remain valid even if the device is shared in a multiprogramming environment.

1.2.2 Mappings

It has already been observed that for reasons of security and because of uncertainty over which of several identical resources will be assigned, programs running in an operating system environment are prevented from directly accessing many of the computer's resources. Consequently, such resources are referenced using programmer-invented symbols rather than physical addresses. The operating system assigns real resources to symbolically named resources, and creates a map from symbolic name space to the space of real resources. Operations on symbolically named devices are interpreted and symbolic device names are mapped to physical device addresses. Inverse maps must also be available so that signals from real devices can be associated with the symbolically named devices.

Tables representing maps between various symbolic name spaces and device address spaces can consume a large fraction of the space occupied by operating systems; many system actions employ these maps or their inverses. For example, in file manipulation maps between external file name space, external volume name space, symbolic file name space, and physical device address space, are all required.

Chapter II

Parallel SETL

In presenting operating system algorithms, it is desirable to focus on algorithmic content rather than machine dependent details. A natural approach is to present programs embodying the algorithms in a higher level language.

The language used should have the property of not forcing artificial structure on the data which the operating system manipulates. While in practice the structure chosen may have a great bearing on performance, this choice of data structure may be hardware dependent and should not be dictated by the language chosen. In the programs to be given in this thesis, the focus will be on algorithmic content. The potential advantage for a structure free notation will become apparent when we come to represent the many maps which an operating system requires. A crisp, mathematical notation preserves spirit of the algorithms manipulating these maps; this spirit would be obscured by a language which insisted upon structural details. Accordingly our algorithms will, after appropriate informal description, be presented as SETL programs. It will be seen to be of particular advantage that SETL allows arbitrary index sets (domains of maps) without requiring explicit attention to how indices map into integers or other preferred entities.

2.1 SETL Deficiencies

There are however several notions needed in describing operating systems which cannot be expressed in ordinary SETL. Operating systems coordinate multiple processes, and mechanisms are required to identify these processes and to specify the way in which control passes between them. Interrupt mechanisms are necessary as a means of communicating between operating system and user programs, and it must be possible to specify protection mechanisms in order to allow programs to run concurrently in a safe way. Other features which must be described in presenting operating system algorithms, but which are not available in standard SETL, include clocks and timers, external device communication, resource allocation, and resource sharing.

To make it possible to describe operating system algorithms, we shall add idealized versions of these features to SETL. Our operating system oriented extensions will not necessarily correspond directly to the hardware of a specific machine; however, all our extensions can be realized on third generation or later computing systems. SETL with operating system extensions will be called PSETL, short for parallel SETL.

Our SETL enhancements will take several forms. Special sets which indicate the state of components of the computing system, such as the process in control of the CPU, will be added to SETL. These sets will be accessible to certain operating system routines but not to user programs. Names of special sets, which are only accessible to the operating system sill appear in boldface type in this text.

A number of new operations will be defined. Of these operations, only a few are truly fundamental; the remainder can be defined in terms of the fundamental ones and ordinary SETL. But most of the time it will be convenient to think in terms of of the macro operations which will be

introduced. Of course, the representation of these macros in terms of a stripped down PSETL embodies some quite fundamental operating system algorithms.

2.2 An Overall View of the Extensions Which Will Define PSETL

2.2.1 Jobs and Processes

The coarsest identification of independent programs and data within the computer will be by job. To unify the control structures of the operating system, the operating system itself will also be considered to be a job, although none of the user jobs are independent of the operating system. With each job, a 'mover' is associated as a means of identification, and a special set, "**movers**", within the operating system will hold the identifiers of all currently active jobs.

In representing the operating system's processing of a job, it is not sufficient to take into account only the code (i.e. program text) and data which comprise the job; the varying data state generated when the program executes must also be considered. The words 'program' and 'procedure' will be reserved to signify the (static) pattern of bits which the hardware is initially given to execute. A program in execution, i.e. a program already coupled to data and thus at least potentially 'in motion' will be called a 'process'. The notion of process can be put formally by mimicking the definition of a computation used in discussing Turing machines. A process is the sequence of states which a CPU takes on in executing a program. Since we wish to allow programs to initiate independent paths of execution (i.e. to initiate parallel processing), we will allow for more than one process to be associated with a job. Each process corresponds to a complete path taken by a CPU through the program.

Formally, a process is then identical with the history of a CPU's execution of a program. In order to be able to regard such a history, (which may actually be executed in bits and pieces) as an identifiable 'thing', we will associate a **unique blank atom** p with each process at the time of its inception; p will serve, and occasionally be referred to, as the **process identifier**, though sometimes in the interests of brevity, we will refer to this identifier simply as 'a process'. That is, we will sometimes use the term 'process' informally, in the sense explained in the previous paragraph. Thus we will use expressions such as 'interrupting a process' to mean that a CPU is diverted to other activities between the execution of successive steps associated with a process, 'starting a process' to mean that the CPU is forced to take on the state indicated by a state vector supplied with the process identifier, and 'resuming a process' (presumably after an interrupt) to mean that some CPU which was interrupted after the nth step of a process is now continuing execution from the n+1st step associated with that process. An operating system is an example of a job using multiple processes, whereas the majority of (today's) applications consist of a single process.

In the discussion which follows, the special set of pairs, **processes**, contains elements of the form <m,p>, where m is a mover and p a process belonging to m. The set **processes**{m} consists of all active processes belonging to the mover m.

Let us first consider the case in which only a single CPU is available. The special variable **CPUcontrol** identifies the process controlling the CPU; the contents of **CPUcontrol** is always a member of the set **processes**. The special set **state**, which is indexed by members of **processes**,

gives the environment for each process. By the environment of a process, we mean all the information necessary to define the path which execution of a process will take when the process comes into control of the CPU. Details of what is specified in an environment are machine and implementation dependent, and may include information concerning the code block to be executed, the next instruction within it to be executed, and the values of all variables accessible to the process, together with the pattern of calls effective at a given moment, etc. For an element s∈**state**, environment(s) extracts the environment part of s, and processpart(s) identifies the process described by s. The macros **loctr**, **code**, and **privilege** extract the components of an environment which give the next instruction to be executed, the string of bits which is the executable code within that environment, and the privilege class associated with that environment.

Process switching is achieved by changing **CPUcontrol**. (See examples 2.2.6.3 and the simple dispatcher in 2.4.3.1 for examples of this, i.e., for process switching by assignment to **CPUcontrol**.) Ordinary 'go-tos' are a particular case of modifications of **state**; more specifically, for a privileged process, the two statements:

go to L; and loctr(**state**(**CPUcontrol**))=L; ,

have the same effect. The first is still the prefered form; the second is shown by way of explanation.

2.2.2 Control of Interrupts

Interruption is a major communication mechanism between parts of an operating system and problem programs. Generally this mechanism has no counterpart in higher level languages, since these languages are intended to describe simple, non-parallel, deterministic algorithms.

In SETL, the standard flow of control is from one statement to the next sequential statement, with the exception of branch statements, if statements, 'while' and '∀' iterations, subprogram calls and returns. An interrupt is an additional control mechanism which causes the flow of control to move to a specific instruction in memory on the occurrence of a special event, such as an end of an I/O operation, a machine malfunction, the end of a measured time interval, or a rare side effect of an instruction. If there are several different statements to which control may flow on interruption, depending on the condition which caused the interrupt, all the conditions which cause control to flow to the same statement will be said to belong to a single 'interrupt class'.

Two features are required to describe an interrupt system. It must be possible to define the process which is invoked on the occurrence of particular interrupts, and it must be possible to inactivate the interrupt system.

We define a set, **interrupt**, which consists of a collection of pairs of the form <int,intprocess>, where int specifies an interrupt class, and 'intprocess' identifies the process invoked when an interrupt of class int is encountered. 'intprocess' must be of an appropriate form to identify a process, as described in 2.2.1. A set **resume** takes on the value which **CPUcontrol** had immediately before the moment of interruption, and can be used to resume the interrupted process, via the simple statement:

CPUcontrol = **resume**;

After an interrupt has occurred further details concerning it are contained in the variable **cause**; the

value of this variable shows all relevant interrupt-related information. Only privileged operating system code has access to the sets **interrupt**, **cause**, and **resume**, which abstractly represent the hardware mechanisms which are directly associated with physical interrupts.

In PSETL, the interrupt system is generally active or enabled, so that interruption is generally possible. At certain moments, however, interruption is intolerable, and computing systems therefore contain instructions for disabling and enabling the interrupt system under program control. A similar mechanism is required for PSETL. However, the PSETL interrupt disabling feature will be less general than that found on most computing systems, in that it will not be possible to keep the computing system permanently disabled. This may cause minor inconveniences in some cases, but it will have the beneficial property of making it linguistically impossible to introduce a "bug" which prevents the system from re-enabling the interrupt system.

To this end we add to SETL a new semantic facility, the **disabled block**, which has the form:

(disable) block; end disable;

The block of code in a disabled block is restricted in the following ways:

1. There may be no **while** iteration headers within the block.
2. Only foward branches within the block are allowed.
3. Branches out of the block end the disabled condition.
4. Calls to user defined subroutines, or subroutine returns, end the disabled condition.

While in the disabled state, the process in control of the CPU is guaranteed uninterrupted control. The restrictions on the disabled block guarantee that a disabled process cannot permanently hold the CPU.

In the case of a multi-CPU configuration, only one CPU may be in the disabled state at a given time. Attempted entry into a disabled block while another CPU is already disabled implies a wait, which is known to be finite because of linguistic limitations on the contents of a disabled block. Thus, in PSETL, disabled blocks may be used to guarantee integrity of special sets during their use.

If the procedure executed as the result of an interrupt begins with a disabled-block, no additional interrupts can occur on that CPU until the end of the disabled-block. Failure to start an interrupt-activated process with a disabled-block would allow a second interrupt to overwrite **cause** and **resume** before the process activated by the first interrupt can save them, thereby preventing proper recovery from the first interrupt.

2.2.3 Private and Shared Data

Conventional SETL distinguishes between two types of variables, locally owned and external. Locally owned variables are those which occur within a subprogram and are not otherwise declared. Locally owned variables can be referenced by name only within the subprogram in which they are defined, although their values may be transmitted between subprograms using the standard SETL 'call' mechanisms. External variables are explicitly declared by use of the SETL **include** and **global** statements. External variables may often be thought of as implicit arguments.

PSETL requires a third class of variable. Recall that the notion of a process involves the further notion of 'path of control of a CPU'. It is possible that several paths of control should execute the

same body of code (though of course at least some parts of their environments would be different). Allowing interruption and multiprocessing raises the possibility that several processes may be executing a common subprogram concurrently. Of the variables referenced within the subprogram, some, for example, may have 'overall' significance to the subprogram itself, whereas others may have 'separate' significance for several processes, more than one of which may be executing the subprogram.

In the first case, we wish only one instance of the variable to exist, regardless of the number of processes concurrently executing the subprogram. An example of such a variable is one which represents the number of processes currently executing the subprogram. Another example is a variable representing a table read by all processes currently executing a subprogram. Such variables will be called **shared variables**.

In the second case, there exist as many instances of a variable as there are processes using the subroutine. Such a variable, for example, can represent the time at which the process entered the subprogram. These variables are in effect **private**. A process using such a variable need not be concerned about possible interaction through that variable with another process. The local variables of SETL will be taken to be ipso facto private variables of PSETL; we will also allow certain SETL global variables to be private.

We adopt the convention that shared variables are to be declared at the beginning of a subprogram by means of the **shared** statement, as follows:

shared v_1, v_2, ..., v_n;

Recognizing that a single subprogram can be executed on behalf of several processes, SETL **initially** blocks will be understood to be entered on the first execution of a subprogram on behalf of each process. Put another way, the mechanism which controls entry into the initial block is private.

2.2.4 Standard Queues and Facilities

In operating systems, it is common to regard work as being queued on an object such as a process, a data structure, or an I/O device. PSETL provides standardized queues, which it relates to a special set, **workset**. It also provides standard mechanisms for adding elements to and deleting elements from these queues. For an object j, **workset**{j} is the queue of work stacked on j. The structure of the queues is immaterial to most of our discussion; suffice it to say that they can be either linked lists or tuples.

With **workset**, we provide several subroutines which allow the workset for an object to be regarded as a queue without reference to the specific structure of the workset. The function readfirst(j) returns the first item on j's workqueue, unless it is empty, in which case it returns Ω. The subroutine remove(j,x) deletes the item x from j's workqueue. Since one frequently wishes to access the first item of a queue and remove it from the queue, a function, getfirst, is provided which can simply be defined by:

```
define getfirst(j);
    remove(j,readfirst(j) is x);
```

```
    return x;
end getfirst;
```

To facilitate searching a workset for an item satisfying a condition, macro findfirst(j,x,C(x)) is provided. The value of this function is the first item x in j's workset satisfying the condition C(x). To augment worksets, we have the subroutine insertafter(j,x,y), which makes y the successor of x in j's workqueue (unless x is not in the workqueue, in which case y becomes the last item). Similarly, insertbefore(j,x,y) makes y the predecessor of x, or the first item if **n**(x∈workset{j}). We also introduce two other useful auxiliary functions by the informal definitions:

putfirst(j,x) = insertbefore(j,readfirst(j),x), and
putlast(j,x) = insertafter(j,Ω,x).

Various important notions connected with the overall concept of dedicated computing system portions will be represented in PSETL using a special set called **facilities**. An object x is a facility if if the test x∈**facilities** is true. The special set **busy** identifies those facilities which are momentarily in use or reserved. The special set **holds** identifies the facilities which are busy on behalf of each process. If p∈**processes**, then **holds**{p} is that subset of **busy** which is dedicated to p.

We also regard the pool of available CPUs as an object with a workset. The workset associated with the pool of CPUs contains all the processes which are ready to start or to continue to execute, but which are not running because every CPU is engaged in other activity. The following line of code may well serve as the final line of a dispatcher (a routine which selects the next process to be executed and starts the CPU on that process):

CPUcontrol = getfirst(CPU);

To ease the coding of the common operating system operation which delays execution of a subprogram until a reserved facility becomes available, a new form of subprogram is added to PSETL. This is the **queued subroutine**. A queued subroutine is defined by a header of the form:

define **qd** name(a_1,...,a_n) **on** fac;

This header is distinguished from the conventional SETL subroutine header by the keywords **qd** and **on** and by the expression following the keyword **on**. A queued subroutine with the above header is entered only when the calling program has exclusive control of the facility fac, which is generally an expression in the arguments a_1, ..., a_n.

Each queued subroutine must use the label "nonexistent" in its body. Control passes to this label in the event that **n**(fac∈**facilities**). If fac∈**facilities**, the subroutine is entered as soon as fac is not busy. At the moment of entry, fac is made busy on behalf of the process invoking the queued subroutine by adding fac to the sets **busy** and **holds**. It is the process's responsibility to release the facility when it is no longer needed by issuing the statement:

free fac;

In addition to the subroutine header shown at the beginning of this section, the various other function definition forms which SETL provides, including infix, postfix and prefix forms, are allowed to have the obvious queued forms, too. Queued subprograms are invoked in the same

manner as conventional subprograms. This frees the caller from concern with many detailed synchronization activities implied by the use of facilities.

2.2.5 Process Control

Among an operating system's prime responsibilities is the control of processes. Functions belonging to this general heading include process creation and termination, process suspension, and interprocess communication. We shall now describe statements useful in supporting these important functions. We point out that the operations described in this section are available only to privileged processes in PSETL.

2.2.5.1 Process Creation

The PSETL statement:

split to s(e) **for** p_1;

is used to begin a new process from the state s; the new process is identified by processpart(s), and execution begins at loctr(environment(s)). The pair $<p_1,e>$ is passed to this process through its environment. The new process can extract the pair $<p_1,e>$ from its environment by applying the positional macro **initialvar** to its environment. Moreover, the positional macros 'ancestor' and 'info' retrieve p_1 and e from initialvar(s). Thus, a process may identify the process which initiated it by retrieving ancestor(initialvar(**state(CPUcontrol)**)), and it may reference the information being passed to it by retrieving info(initialvar(**state(CPUcontrol)**)).

2.2.5.2 Process Suspension

A privileged process may suspend its own operation until a specified condition is met. The PSETL statement:

await cond;

causes the process which issued the **await** to test the condition cond, and if it is found to be false, to suspend operation until cond becomes true. It is clear that for the condition to change in value other processes must be able to proceed during the suspension of the process which issued the await. (Non-privileged programs will be provided with a similar capability in the form of an operating system service which is invoked by a standard operating system request.)

Processes suspended by **await** statements will be saved in the special set **waitset**. When a process x is entered into **waitset**, loctr(**state**(x)) is set up to re-evaluate the condition cond.

2.2.5.3 Interprocess Communication

A process may require the services of a second process, even though in many cases the time at which the services are rendered are not material to the first process, which moves forward as soon as the parameters for the second process are transmitted. The second process, on the other hand, may already be occupied with another request. A PSETL statement, **enqueue** provides this linkage by using the workset for the second process. The PSETL statement:

enqueue e **on** p_2 **for** p_1;

enters the pair $<p_1,e>$ on p_2's workqueue. The process p_2 must be written to examine its workqueue for additional requests each time it finishes servicing a request. See example 2.2.6.4.

2.2.5.4 Process Termination

A process can terminate its execution by executing the PSETL statement:

term;

This causes all facilities held by the process to be **free**'d, and its workqueue to be purged.

A process can force the termination of a second process by executing the PSETL statment:

kill p_2;

Generally, the issuing process must have at least as high a level of privilege as the process it kills. As on the execution of a **term** statement, the **kill**'d process's workqueue is purged, and facilities held by it are **free**'d.

2.2.6 Examples

2.2.6.1 The following trivial routine, which we will use frequently in this work, can be called to delay a process until a facility x can be secured:

```
define qd reserve(x) on x;
nonexistent: return;
end;
```

2.2.6.2 Dijkstra [Di68] defines P and V operations for process synchronization using semaphores, which are initialized to 0 or 1.

> "A process, Q say, that performs the operation 'P(sem)' decreases the value of the semaphore called 'sem' by 1. If the resultant value of the semaphore concerned is non-negative, process Q can continue with the execution of its next statement; if, however, the resulting value is negative, process Q is stopped and booked on a waiting list associated with the semaphore concerned. Until further notice (i.e. a V operation on this very same semaphore), dynamic progress of Q is not logically possible...
>
> "A process, 'R' say, that performs the operation 'V(sem)' increases the value of the semaphore called 'sem' by 1. If the resulting value of the semaphore concerned is positive, the V-operation has no further effect; if, however, the resulting value of the semaphore concerned is non-positive, one of the processes booked on its waiting list is removed from this waiting list, i.e. its dynamic progress is again logically possible."

In PSETL, with the understanding that semaphore variables are facilities, that semaphores initialized to 0 are **busy**, and that semval is a map from semaphores to their values, we can express the P and V operations by:

```
define P(sem); shared semval;
    (disable) semval(sem)=semval(sem)-1 is news;
        if news ge 0 then sem in busy;
```

```
            else reserve(sem);
        end if;
    end disable;
end P;

define V(sem); shared semval;
    (disable) semval(sem)=semval(sem)+1 is news;
        if news le 0 then free sem;;
    end disable;
end V;
```

Clearly, if one merely desires to synchronize processes, without requiring that a count of delayed processes be kept explicitly by sem, our dictions are rich enough to allow 'reserve(sem);' for 'P(sem)' and 'free(sem);' for 'V(sem)'. The number of delayed processes can always be computed by #workset{sem}.

2.2.6.3 A more complex example: Let d be a set all of whose elements are facilities. If all elements of d must be secured before a process can continue, one can simply insert the code:

```
(∀fac∈d) reserve(fac);;
```

at an appropriate position. The above code achieves reservations one at a time. On the other hand, it may be preferable to seize each device as soon as it becomes available, since if one follows any particular sequential order, devices available at the start of the sequence but required later may be preempted by another process by the time an attempt is made to reserve them. A parallel reservation strategy must surely be at least as fast as the sequential approach, and may be written in PSETL as follows:

```
    w=newat; x=state(CPUcontrol); loctr(x)=S;
    (∀fac∈d) split to <<w,newat>,x>(fac) for CPUcontrol;;
    await #processes{w}eq 0;
      .
      .
      .
S: reserve(info(initialvar(state(CPUcontrol)) is i));
    (disable)
        i in holds;
        <CPUcontrol,info(i)> out holds;
        term;
    end disable;
```

Recall that initialvar(**state(CPUcontrol)**) is a pair <p,fac>, where p is the process which spawned the reservation processes, and fac is the facility to be reserved. The statement S reserves the facility fac on behalf of the process q executing S; the following disabled block switches the reservation to the process p to avoid the reservation being lost when **term** is executed by q.

2.2.6.4 A final example: Let us sketch a simple output routine which accepts a single string of characters as input and prints the string using embedded **er** characters to deduce where the lines start. We assume that the routine is invoked by:

```
enqueue str on printer for CPUcontrol;
```

where printer identifies the process associated with the program to be given below. The advantage

to the calling program is that it can proceed immediately after the **enqueue** regardless of the work already scheduled for the printer or the time physically required to print the string.

```
output: await #workset{printer} ne 0;
    str=info(getfirst(printer));
    j=1;
    (while (j≤∃k≤#str | str(k) eq er) doing j=k+1;)
        printout str(j:k-j);
    end while;
    go to output;
```

The first statement causes 'output' to wait until (or unless) there is work stacked on its workqueue, and the second statement extracts the next string to be printed from the workqueue. The remainder of the code shown above is straightfoward SETL; we assume that 'printout' is a more primitive routine which prints its argument on a new line, left adjusted, on a printer.

2.3 A Remark Concerning Machine Dependent Features and PSETL

The PSETL features introduced in section 2.2 allow the desription of a good portion of operating systems. At some stage, however, we will wish to stop hiding crucial underlying details by linguistic facades, and to face them. Some (but not all) of these underlying details are machine dependent. Those which are not we may subsequently wish to describe in additional detail; of course details which are highly machine dependent we exclude as belonging to a different type of discussion. Thus, for example, a **read** verb in PSETL describes an input action, and presumably is translated into a call on a standard I/O package. If we wish to describe the I/O package in PSETL, we are ultimately faced with the necessity of issuing I/O instructions which carry out the **read**, a task which cannot be circumvented by using another PSETL read. Such ultimate levels of machine dependence can only be handled by the use of primitive machine-level subprograms or by special bit patterns or other data objects whose significance must be described in English and coded in a lower level language.

The special sets, **interrupt** and **cause**, are additional examples of features of PSETL whose inner details are so machine dependent that detailed definition is left to the actual system implementer. In our PSETL discussion we may assume certain distinct interrupt classes, and some particular manner in which the information describing the circumstances of the interrupt is posted, but we shall not describe the machine-level mechanisms which cause this to occur.

2.3.1 Time and Clocks

We assume the existence of two special global integer variables, **timer** and **clock**. The variable **clock** is incremented by one automatically by the hardware every n microseconds, where n is a hardware dependent parameter. We further specify that **clock** is **read-only**, i.e., that storage operators applied to **clock** have no effect. By reading a message from the operator's console giving the real time, **clock** can then be used to compute the real time at later points in time.

The variable **timer** may be accessed by fetch or store operations. Whenever the **clock** changes value and matches (or exceeds) the value in **timer**, a **time interrupt** occurs.

2.3.2 Input/Output

The SETL input/output statement forms should be viewed as subprogram calls, for the "statements" (or instructions) which perform the physical input/output operations are privileged instructions. Even the subprograms which interpret the SETL I/O cannot use the physical I/O instructions of the computer directly; they must use the services of the operating system to interpret more primitive I/O operations than the ones supplied by SETL.

A sample set of such operating system services in described in section 3.3.3, and used in all the sample operating systems of Chapters III, IV, and V. In addition, the uniprogrammed operating system of section 3.5.3 shows in detail how the operating system interprets I/O.

When the operating system interprets the requests for I/O, it must eventually specify physical I/O operations. These operations are highly machine dependent, and often require observing timing constraints. We therefore do not attempt to specify in detail how the physical I/O instructions should be represented in PSETL; however, for the sake of describing that portion of the operating system which interprets input/output requests, we will use the following statement formats to specify physical I/O (which requires no further interpretation except by the hardware):

1. Read operations are coded:
   read(d,command);
   where d spcifies a physical I/O device, and 'command' is a structure which specifies the location and size of the block of main memory to be read into.

2. Write operations are coded:
   write(d,command);

3. Simple control operations are coded
   op(d);
   where d is a device, and 'op' is an operation such as rewind, or backspace. For positioning operations, such as disc-seeks, we write:
   seek(d,c);
   where c specified the cylinder position to which the read/write head is to be moved.

The reader disturbed by the vagueness with which physical I/O is treated here may be consoled by the fact that physical I/O will not be used extensively in the operating system examples which follow in later chapters. With the exception of the sections of code identified as "Monitor Services" at the end of the code of section 3.5.3, the operating system itself uses either SETL I/O, or the monitor services described in section 3.3.1.

2.4 Detailed Account of the Elements of PSETL

In this section, the features of PSETL summarised in section 2.2 are described in detail. Our description is arranged into three headings: **special sets, new primitive operations**, and **macro operations**. In the case of macro operations, possible expansions in terms of SETL using the special sets and new primitive operations are given in order to illuminate the mechanisms involved, although the actual implementation is partly immaterial, since such macros are designed to be thought of as primitive.

In giving prototypes of PSETL statements, we will use symbols in the following standardized ways:

fac represents a facility,
$p,p_1,p_2,...$ represent elements of **processes**,
m represents an element of **movers**,
n identifies a path of control for a mover,
$s,s_1,...$ represent states,
$i,i_1,...$ represent interrupt classes,
$j,j_1,...$ represent system objects having worksets,
$a_1,..., a_n$ represent arguments to subprograms or processes,
$v_1,..., v_n$ represent names of variables,
L and M represent compiler generated labels.

The table which follows shows where the description of each PSETL feature will be found:

2.4.1 Special Sets

2.4.1.1 movers

Elements: SETL 'blank atoms', used as identifiers for independent jobs.
Uses: To identify independent jobs.

2.4.1.2 processes

Elements: pairs of the form <m,n> where m∈ **movers**, and n identifies a process for the mover m.
Uses: For m∈ **movers**, **processes**{m} identifies the set of processes associated with the mover m.

The elements of **processes** are the processes in progress under control of the operating system.

2.4.1.3 state

Elements: pairs of the form <p,e> where p∈ **processes**, and e is the environment associated with the process p.

Uses: For p∈ **processes**, **state**(p) determines all the control information, or environment, concern-

ing the subroutine to execute, the location within this subroutine at which to begin execution, and the values of all variables accessible to the process; information which fully determines the future course of the process p.

2.4.1.4 CPUcpntrol

Uses: **CPUcontrol** identifies the process currently controlling the CPU. A running privileged process can always identify itself by reference to **CPUcontrol**, and can access its environment as **state(CPUcontrol)**.

2.4.1.5 interrupt

Elements: pairs of the form <i,s>
Uses: If the pair <i,s> ϵ **interrupt**, and an interrupt of class i occurs, then **CPUcontrol** is set to p, the former contents of **CPUcontrol** saved in **resume**, and the variable **cause** comes to represent all relevant information concerning the interrupt that has just occurred. To operate correctly, **state**(p) must have been initialized to indicate system privilege, and the first statement executed by the process should be a disabled block.

2.4.1.6 resume

Uses: **resume** has the same structure as **CPUcontrol**, and the same positional macros apply to it. On the occurence of an interrupt, **resume** saves the contents of **CPUcontrol** as it existed immediately before the interruption. To restore an interrupted process immediately, execute:

CPUcontrol=resume;

Otherwise, save the value of **resume** in some appropriate way.

2.4.1.7 cause

Elements: machine dependent
Uses: Makes available, in some suitable form, a 'message' giving additional information regarding an interrupt being processed. For example, on arithmetic exceptions, it might be used to distinguish between overflow, underflow or division by zero; on input/output interrupts, it might indicate the hardware address of the device causing interruption, or whether an attempted I/O operation was successful, and if not, the reasons for failure.

2.4.1.8 workset

Elements: pairs of the form <j,q>, where j is a system object, and q is a queue.
Uses: For an object j, **workset**{j} is the queue of items stacked on j. The structure of the queue elements are dependent on the object j. Note that knowledge of the detailed structure of the queue **workset**{j} is not needed in our algorithms, since subroutines described in 2.4.3.7 perform the queue manipulations by using auxiliary sets the details of which need not concern us at this level of discussion. It is sufficient to know that FIFO order of workqueues is preserved.

2.4.1.9 waitset

Elements: A collection of processes suspended awaiting a condition to be satisfied.
Uses: For x∈**waitset**, executing **CPUcontrol**=x; re-evaluates the awaited condition for the process x, which after this evaluation will either begin to move forward, or will resume its wait.

2.4.1.10 facilities

Elements: Serially re-usable system objects, such as devices, variables or subprograms.
Uses: The elements of **facilities** may be reserved by processes, via use of queued subprograms. If x∈**facilities** and x∈**busy**, then x has been reserved by a process. For objects in **facilities**, the use of queued subroutines and the **free** statement provides automatic management of the objects' worksets, and synchronization of processes with the availability of facilities designated in the queued subprogram header.

2.4.1.11 busy

Elements: members of **facilities**
Uses: x∈**busy** implies that some process is using the facility x.

2.4.1.12 holds

Elements: pairs of the form <p,fac> with p∈**processes**, and fac∈**facilities**.
Uses: <p,fac>∈**holds** implies that the facility fac is busy on behalf of the process p.

2.4.1.13 clock and timer

Elements: Each of these variables is an integer.
Description: **clock** is automatically incremented by 1 every n microseconds by the hardware (where n is a machine dependent constant). A timer interrupt is generated by the hardware whenever the clock becomes zero or **clock** changes value and matches or exceeds **timer**.
Uses: By using input from an external source, one can correlate a value of the **clock** with real time, and thereafter use **clock** to determine by program the time in the outside world. **timer** can be set to cause an interrupt at a predetermined time.

2.4.2 Primitive Operations

2.4.2.1 disabled block

Statement form: (disable) block;
Description: While executing 'block', the interrupt mechanism is disabled. If there are multiple CPUs, only one may be disabled at a time; indeed, the entry of one CPU into a disable block temporarily suspends the activity of all other CPUs which attempt to enter disabled blocks. While-iteration headers and backward branches are syntactic errors within a disabled block. Branches out of the block, returns, or invocations of user defined subprograms end the disabled condition.

2.4.2.2 initial block

Statement form: **initially** block;
Description: On each process's first entry to a subprogram, the 'block' is executed. To effectively execute the initial block only once regardless of the number of unique processes executing the subprogram, use a shared variable to distinguish between a first and subsequent uses of the subprogram.

2.4.2.3 process switching; the special variable 'CPUcontrol'

A CPU is directed to switch processes by assignment to the special variable **CPUcontrol**. The assignment **CPUcontrol**=s; causes the process s to control the CPU starting at loctr(**state**(s)), and to operate in the privilege class privilege(**state**(s)). If a process p relinquishes control of the CPU as follows:

```
    CPUcontrol=x;
L:
```

then if another process issues **CPUcontrol**=p;, then p resumes execution at the statement labelled L in the above program fragment.

2.4.2.4 shared variables

Statement form: **share** $v_1, v_2, ..., v_n$;
Description: Variables declared in a share statement and owned by a subprogram have only one instance in storage, regardless of the number of distinct processes executing the subprogram. Such variables, especially if global, may be used to communicate between subprocesses. For variables not declared as being shared, a unique value exists for each process executing the subprogram.

Macro Operations

2.4.3.1 await

Statement form: **await** cond;
Description: The privileged process issuing an await continues execution if the boolean expression cond is true; otherwise its execution is suspended until cond becomes true.

Expansion:

```
if not cond then
    (disable)
        CPUcontrol in waitset;
        CPUcontrol=dispatcher;
    end disable;
    (disable)
        isok=cond;
        CPUcontrol=dispatcher;
    end disable;
end if;
```

The expansion works in cooperation with the operating system's dispatching process, which, in the above code is identified by the name 'dispatcher'. The variable 'isok' transmits to the dispatching process the recomputed condition. A simple dispatcher is:

```
getwork: waitcopy=waitset;
loop: if waitcopy ne nl
    then s from waitcopy;
        L=loctr(state(s)); /*recompute condition*/
        CPUcontrol=s;
        if isok then s out waitset;
            putlast(CPU,s);
        else loctr(state(s))=L; end if;
            go to loop;
        end if;
    /*if waitcopy eq nl then*/
    s=getfirst(CPU);
    if s eq Ω then go to getwork;;
    CPUcontrol=s; /*give control of CPU to chosen process*/
    go to getwork;
```

Notice in the fifth line that executing **CPUcontrol**=s; causes control to flow to loctr(**state**(s)), which is where the condition cond is recomputed in the above expansion for **await**. If the condition is still not satisfied, loctr(**state**(s)) is reset to recompute cond. Otherwise the process s will proceed beyond the **await** when it next receives control of the CPU. After the dispatcher tests all awaited conditions, and moves processes with satisfied conditions to the CPU's workqueue, The first element of the CPU's workqueue is selected as the next process to run, unless the CPU workqueue is empty, in which case the dispatcher re-examines the unsatisfied conditions. Since the dispatcher is enabled, interrupt response is possible and may result in one of the conditions becoming satisfied.

One may wonder whether a dispatcher can run enabled? The sample dispatcher will work correctly together with the above expansion of **await**. An interrupt during the running of the dispatcher will be handled, but a process which enters a wait condition as the result of such an interrupt having been processed will not be considered for resuming operation until the next time the dispatcher executes the statement at 'getwork'. On the other hand, in handling certain interrupts, an urgent process may be put at the head of the CPU's workqueue, and thus be dispatched before any of the processes which were under consideration at the time that the dispatcher was started.

2.4.3.2 queued subprogram header

Statement forms:

(1) define **qd** name(a_1, ..., a_n) **on** fac;

(2) definef **qd** name(a_1, ..., a_n) **on** fac;

(3) infix, prefix and postfix forms of the above

Description: Entry to the subprogram 'name' is completed only when the facility fac is available. While fac is busy for another process, the calling process is queued on fac. When control reaches the first user-coded statement in the subprogram, facϵ**busy** and <**CPUcontrol**,fac>ϵ**holds**. It is

the responsibility of the calling process to eventually release the facility when no longer needed. A queued subprogram must include a statement labeled 'nonexistent' to which control will flow in the event that fac is not a facility.

Expansion:

```
define name(a1, ..., an);
if n(fac ∈ facilities) then go to nonexistent;;
(disable)
    if fac ∈ busy then
        putlast(fac,CPUcontrol);
        CPUcontrol=dispatcher;
    else
        fac in busy;
        <CPUcontrol,fac> in holds;
    end if;
end disable;
end if;
```

Note: The else-clause is necessary to make fac busy to other processes. Remember that the queued subroutine performs control functions which ordinarily are assumed by the user to be performed by an operating system; the queued subroutine is used to construct an operating system.

The 'reserve' subprogram of section 2.2.6.1 is an example of a queued subprogram. It is used to delay a process until it's argument, a facility, is not busy and can be reserved.

2.4.3.3 free

Statement form: **free** fac;

Description: The facility fac is released by the process which issued the free. fac is removed from **busy** unless another process is enqueued on fac, in which case that process is activated and the facility reserved for that process.

Expansion:

```
(disable)
    v=getfirst(fac);
    <CPUcontrol,fac> out holds;
    if v=Ω then
        fac out busy;
    else
        putlast(CPU,v);
          <v,fac> in holds;
    end if;
end disable;
```

Note: The above expansion assumes that the workqueue for fac is empty if fac is not busy. To guard against other code violating this assumption, the 'else block' above should include the statement: fac **in busy**;.

2.4.3.4 split

Statement form: **split to** s(e) **for** p_1;

Description: In the above statement, s is a state variable, such that processpart(s) represents a new process, p. A new process is created as follows: p is added to **processes**, the pair $<p_1,e>$ is stored in p's environment in such a manner that it can be retrieved by initialvar(environment(s)), s is added to the set **state**, and an entry is made on the CPU's workqueue, indicating that the process p is ready to use the CPU.

Expansion:

```
(disable)
    moverpart(processpart(s) is p) in movers;
    p in processes;
    s in state;
    initialvar(state(p))=<p1,e>;
    putlast(CPU,p);
end disable;
```

The first of the above statements will be a "no-operation" whenever a new process is being created for an already existing mover. When the process being initiated belongs to a new mover, the first statement insures that the new mover is a member of **movers**.

2.4.3.5 enqueue

Statement form: **enqueue** e **on** p **for** p_1;

Description: The pair $<p_1,e>$ is placed at the end of p's workqueue. Upon adding the pair to p's workqueue, the process which executed the enqueue is free to continue execution. A process which services enqueued requests will, upon becoming idle, generally suspend its operation for later resumption by waiting for its workqueue to become non-empty. See example 2.2.6.4.

Expansion: putlast(p,$<p_1,e>$);

2.4.3.6 process termination

Statement forms:

kill p;

term;

Description: The process identified by p is terminated; items already stacked by it on other workqueues are eliminated, facilities held by it are released, and its workqueue is dropped. The statement **term**; is equivalent to **kill CPUcontrol**; and is used by a process to terminate its own execution.

Recall that elements in workset are of the form <s,p,r>, where s identifies the item on whose workqueue the element belongs, p represents the process which has enqueued the request in s's workqueue, and r are the arguments being passed by p to s. Thus, items in workset{s} are of the form <p,r> and ancestor(y) for $y \in$ workset{s} is a process identifier. Thus, the first statement

removes all items enqueued by the process being killed, whereas the third statement, (workset{p}=Ω;), destroys the items enqueued for the process being killed. A possible variant to this expansion would recursively kill the processes which had enqueued work on processes being killed.

Expansion:

```
(disable)
    (∀objects∈ hd[workset], ∀procs∈ workset{objects} | ancestor(procs) eq p)
        remove(objects,procs);
    end ∀x;
    (∀fac∈ holds{p}) free fac;;
    workset{p}=Ω;
    p out processes;
    if #processes{moverpart(p)} eq 0 then
        moverpart(p) out movers;;
    if processpart(state) eq p then
            /*This is the case when term is being executed. Having removed all objects from
            queues associated with the terminating process, the process removes itself from the
            state set, and gives control to the dispatcher.*/
        <CPUcontrol, state(CPUcontrol)>=<dispatcher,nl>;
/*  else
        control continues to the next sequential statement.*/
    end if;
end disable;
```

2.4.3.7 queue management subprograms and macros

Statement forms:

```
findfirst(j,x,C(x));
getfirst(j);
insertafter(j,x,y);
insertbefore(j,x,y);
putfirst(j,x);
putlast(j,x)
readfirst(j)
remove(j,x)
```

Description: The function readfirst(j) returns the first element on object j's workqueue. If there are no elements on the workqueue, then readfirst(j)=Ω. remove(j,x) will remove x from j's workqueue if x is present, in such a manner that the FIFO ordering of the remaining enqueued items is preserved. The function getfirst(j) combines the actions of readfirst and remove, by returning the first item on j's workqueue and removing the item from the queue.

The subroutine insertafter(j,x,y) makes y the successor of x in j's workqueue if x is present; otherwise y becomes the last item in the workqueue. Similarly, insertbefore(j,x,y) makes y the predecessor of x or the first item in the queue. The special cases of adding to either end of a workqueue are handled by putfirst and putlast, which can be defined by:

```
define putfirst(j,x); insertbefore(j,readfirst(j),x); return;;
define putlast(j,x); insertafter(j,newat,x); return;;
```

The macro findfirst(j,x,C(x)) sets x equal to the first item in j's workqueue satisfying the condition C(x).

These subprograms have routine SETL expansions defined by whatever logical structure is chosen for the workqueues. The only PSETL consideration that arises is that these subprograms will have to be be disabled to prevent other processes from modifying the workset while getfirst, putlast, and remove are in operation.

2.4.3.8 positional macros

Forms:

processpart(s)
environment(s)
privilege(en)
loctr(en)
code(en)
initialvar(en)
ancestor(x)
info(x)
moverpart(p)

Description: Once a specific structure for **state** has been chosen, the first two macros, to be used on objects with the same structure as **state**, extract the process portion and environment portion of their arguments, respectively. If for example, **state** is a pair, we could use the conventions processpart(s)=s(1), and environment(s)=s(2).

privilege, loctr, code, and initialvar apply to objects with the same structure as an environment. If en has the structure of an environment, then privilege(en) extracts the privilege portion of en, loctr(en) extracts the location portion of en, and initialvar(en) extracts the pair <p,e> from en, where p is the process which initiated the process having en as an environment, and e is initialization information passed by p to that process. code(en) extracts the string of bits which is the executable code within the environment en.

ancestor(x) and info(x) apply to objects of the form occuring on workqueues, and respectively reference the process which placed the object on the workqueue, and the request being transmitted through the workqueue. These macros are also applicable to objects retrieved by the initialvar macro.

moverpart(p) extracts, from a process identifier p, the identification of the mover to which it belongs.

2.5 Other Proposals for High Level Language Operating System Primitves

Dijkstra [Di65], Hoare [Hoa], and Brinch Hansen [B] propose several dictions for coordinating parallel processes; all of these dictions are included in Brinch Hansen's book. A comparison of these dictions with those of PSETL follows.

The notation [Hoa, b].

cobegin S_1; S_2; ... S_n **coend**

is used to indicate that the statements S_1, S_2, ..., S_n may be executed concurrently (or in any order). A simple condition for the results of the computations in a parallel block to be independent of the order of their execution is that the statements S_1, S_2, ..., S_n be disjoint, that is, no S_i may change a variable referenced by an S_j, $j \neq i$. Restrictions on the nature of the S_i [Hoa] permit a compiler to check that the S_i are disjoint. The **cobegin** block gives each parallel statement the same environment. PSETL's **split** statement provides similar capability. In PSETL, each process's environment must be specified before the process can be initiated, and these environments are generally different. No effort was made in PSETL to make the disjointness of parallel processes decidable by the compiler.

To relax the disjointness requirement of **cobegin**, mutual exclusion is provided by the diction [Hoa, B]:

region v **do** S

The block S is called a critical region on the variable v. Only one critical region on any variable may be executed at a time. The compiler can recognize references to v outside a critical region, and treat such references as errors. One requirement placed on critical regions is that control leave the critical region within a finite time, although no means of enforcing that requirement are described in the above-mentioned references.

We can construct the critical region from our PSETL primitives. If we do not put additional requirements on the PSETL compiler, the detection of references to the critical variable v outside the region would be dynamic. In PSETL, the diction **region** v becomes 'reserve (v);'. The variable v itself is implemented as a function, which operates on a variable not directly accessible to v's callers, e.g., an internal structure in v represents v.

```
definef v(args); /*Internal variable var represents v*/
if v∈holds{CPUcontrol} then
        /*Valid reference.*/
    (load) return var(args);;
    (store) var(args)=result;;
else
    /*Invalid reference -- outside 'critical region'*/
    error;  /*Some kind of diagnostic action*/
end if;
end v;
```

At the end of the critical region, we need: '**free** v;'.

A conditional critical region [Hoa, B] is a critical region which contains an **await** similar to the PSETL **await**, with the difference that if the condition is not met, the critical variable becomes available, and the delayed process re-enters the critical region at the end of the **await**. In PSETL, the code:

```
R: reserve(v);
   .
   .
   .
   if n B then
```

```
        free v;
        v in holds{CPUcontrol}; /*This statement will make it possilbe for the condition B in
            the next statement to be evaluated. B involves the "variable" v, in the manner
            described in the discussion above on critical regions, and unless
            v∈holds{CPUcontrol}, the reference to v (c.f. above) would be invalid. Yet having
            free'd v, it is available to other processes. Once B holds true, the reservation must
            must be reestablished.*/
        await B;
        v out holds{CPUcontrol};
        go to R;
    end if;
    .
    .
    .
    free v;
```

gives the same effect as Brinch Hansen's:

region v **do begin** . . . **await** B . . . **end**

PSETL's **workset** can be used to create 'message buffers' [MMc, B] of a kind described by Brinch-Hansen. Our **worksets** are not specified to be of a-priori limited capacity; getfirst, putlast, etc, would have to take the finiteness into account by using **awaits** in the event that the workqueues were full.

The general semaphore [Di65] has already been shown in example 2.2.6.2 to be realizable in PSETL. For binary semaphores, the P operation is simply the PSETL 'reserve(sem);', and the V operation, '**free** sem;'.

Brinch Hansen observes that each of the synchronizing mechanisms, semaphores, message buffers, critical regions, conditional critical regions, and event queues can be realized in terms of any of the other mechanisms of this list; however, it is only more convenient and natural to use the various mechanisms in different situations. Thus, once the semaphores were shown to be expressible in PSETL, it is not surprising that the other synchronizing dictions could be realized in PSETL, too.

Guarantees that a V(sem) will be issued, or that a critical region will terminate execution are lacking, although their need is acknowledged [Hoa, B]. The critical region, (except for its lack of guaranteed finite execution) is an easy and natural diction to use in a structured programming language. There is also some danger of deadlock with nested critical regions.

The PSETL primitives were chosen to be closer to capabilities of contemporary computers. The aim was to specify in PSETL process synchronizing dictions. Thus, the operating system designer can specify in detail in PSETL, how the synchronizing mechanisms are to be realized on his machine.

In the short examples given in the texts of [B], [Hoa], and [Di65], the use of parallel processing dictions are easily comprehended. However, no large body of code comparable to the systems in Chapter III and Chapter V are given; thus one cannot compare the readability of PSETL with, say, Pascal [W71] augmented by parallel processing primitives.

Donovan and Madnick [DM] do present a complete operating system in code, although it is written in assembly language for the IBM System/370. The Dijkstra semaphore is used as the synchronization mechanism in this system. In the code given by Donovan and Madnick, the P and V operations are physically close enough so that the reader may easily convince himself that the critical sections between these operations have finite execution time. The code suffers from the usual assembly language difficulties: opaqueness of data structures, a mass of details imposed by the language (in particular, the appearance of the USING pseudo-instruction, whose correctness is difficult to ascertain), etc.

Chapter III

A Simple Operating System

In this chapter, an operating systems will be presented in a fair amount of detail to illustrate various control mechanisms and mappings commonly used in operating systems, and to make vivid the many detailed design alternatives which arise and must be resolved before an operating system can actually be implemented. Some machine dependent details will enter; care will be taken to distinguish between these and operating system aspects which are machine independent.

3.1 System Objectives

The operating system to be presented here is intended to run on a single CPU configuration operating in a non-interactive environment in a uniprogramming mode. The system to be presented will typify many of the more advanced 'second generation' systems, as well as small, less ambitious 'third generation' systems. In the context of this discussion, 'non-interactive' will be taken to mean that the system allows the running of jobs submitted by users at the computer installation, but only without assistance from the user during execution. This means that all the data for the run is available when the job is submitted and that the only operational decisions which the user can make are those which he has preprogrammed into his programs or which he can express in a **job control language**.

'Uniprogramming' will be taken to imply that only one user run is in execution at a time, and that a user job runs to completion without the CPU being temporarily diverted to other user jobs. By 'user job', we mean just the code submitted with the job. Indeed, there will be housekeeping to be performed for each job before and after its execution, and this housekeeping will be multiprogrammed with the running of other jobs. Note that a designer may be constrained by a small main memory to implement a uniprogrammed design. Indeed severe memory limitation can make it difficult to spare the extra space needed for multiprogrammed control, and can also make it unlikely that several jobs will fit into main memory concurrently.

The constraint to uniprogramming which we begin by assuming limits the degree to which hardware utilization can be optimized. A list of reasonable objectives for a uniprogrammed system are: to minimize job-transition times, to allow jobs to be assigned priorities which influence the scheduling scheme, and to provide library facilities for programs and data. Jobs will be processed in the order in which they are submitted, perhaps within priority classes. The only exception to this FIFO scheme will be to utilize device mounting time, during which the highest priority available job whose estimated running time is less than the expected mounting time, is run.

Users will be allowed and required to reference input/output devices symbolically. Several advantages result from such addressing. If several devices of a given type are provided with the computing system, a job which requires a subset of these devices can run whenever a sufficient number of devices are available; with absolute referencing, a specific subset must be available. In the case of temporary files, a variety of devices may be apropos, and symbolic addressing permits the job to run if any such device is available. Furthermore, premounting of devices while a job

using the same class of devices is running is made possible by the flexibility of symbolic referencing.

From the users' point of view, a simple job control mechanism is an important objective. When the computing system is used in perfectly typical ways, minimum control specification should be required from the user. To this end, default values should be established for most job description parameters, moreover, it should be possible for each user to establish the default values tailored to his personal needs.

3.2 Job Control Language[1]

Job control language is used to describe various aspects of a job to the operating system, such as the job's estimated running time, the files it requires, the partitioning of the job into steps, and the dependence of a job step on previous steps. Four types of control statements will be used by our operating system: a job statement to identify a job, a job-step statement to identify subsections of a job, a data file statement to identify and describe files needed by the various job steps, and an end-of-data statement to mark the end of data files which are included with the job. These job control statements are part of a language in the sense that they allow communication of the operational characteristics of a job which would have to be given to an operator if there were no operating system.

Job control language statements, or JCL for short, are imbedded in the statements and data which comprise the job. The general structure of a job consists of a job statement, followed by several job steps. Each job step consists of a job step statement, data file statements, data, and an end-of-data statement.

Each JCL statement other than an end-of-data statement will be represented by a SETL set (which we will use as a mapping):

$$\{\langle\text{'label'},st\rangle,\langle\text{'command'},c\rangle,\langle s_1,p_1\rangle,\langle s_2,p_2\rangle,\ldots\langle s_n,p_n\rangle\}$$

where st is a string representing the statment's label, c is a string identifying the JCL statement type, $s_1,\ldots,s_n$ are names of subfields, and $p_1,\ldots,p_n$ are parameters associated with the subfields $s_1,\ldots,s_n$.

With the exception of the end-of-data statement, each type of JCL statement contains several subfields which convey information. Many of these subfields are optional, in that they are not mandatory in every use of the JCL statement. To make the identification of subfields easy, each subfield will have the form:

<sname,info>

---

[1] The job control language for this operating system is patterned after the job control language for IBM OS/360 [I72].

where sname is the name of the subfield and info is the information being transmitted in that subfield. For example, to estimate 5 minutes running time on a job statement, the subfield

<'time',5>

would appear in the statement. Because the subfields are identified by name, their order in a JCL statement is immaterial.

Descriptions of how to code the various JCL statements follow. These detailed descriptions will in fact describe to a great extent the services which this operating system offers its users.

3.2.1 The Job Statement

The form of the job statement is:

{<'label',nm>,<'command','JOB'>,a_1,...,a_n}

In the above prototype, nm is a name attched to the job, and a_1,...,a_n are n subfields. The subfields which we allow will be representative of the information which can be transmitted in the job statement in most systems, although the more elaborate systems allow for a larger variety of specifications to be given.

The NAME Subfield

This parameter is used to identify the user responsible for the job. It must match one of the names of valid users stored in the operating system. The name parameter is the only required parameter on the job statement. A comma or a blank ends the name subfield. An example of a valid job statement is:

{<'label','XYZ'>,<'command','JOB'>,<'name','MARKSTEIN'>}

The TIME Subfield

This parameter gives the estimated CPU running time for the job. If this time elapses during execution, the operating system will automatically terminate the job. The TIME specification is given in units of minutes:

<'TIME',x>

where x is a real number representing the number of minutes estimated for the job's running time.

The PRIORITY Subfield

This parameter is used to determine in which priority class the job is to be placed for scheduling of the CPU. There are 10 classes, denoted by the integers 0 through 9, class 9 having the highest priority. If not specified, the priority class is taken from the user profile; if specified, the priority

used is the lesser of the specified priority and the maximum allowable priority for the user as given in the user profile. A sample use is:

{<'label','QUICK'>,<'command','JOB'>,<'name','VIP'>,<'priority',9>};

3.2.2 The Job-Step Statement

The form of the jobstep statement is

{<'label',nm>,<'command','EXEC'>,a_1,...,a_n}

In the above prototype, nm is a name attached to the jobstep. The following subfields are recognised:

The PROG Subfield

This subfield identifies the data file which contains the program to be executed. If the data file is already catalogued by the system, no further information is required; otherwise, a data file statement for the file must be given in the JCL for this job step. An example of a valid job step statement is:

{<'label','A'>,<'command','EXEC'>,<'PROG','ALPHA'>}

which causes file alpha to be loaded and run.

If several files are to be loaded, their names are given as a tuple, e.g.

{<'label','B'>,<'command','EXEC'>,<'PROG',<'ALPHA','BETA'>>}

The PROC Subfield

This subfield identifies a data file which contains JCL for one or more job steps to be executed. Use of PROCs (short for procedures) enables libraries of JCL to be built up and reduces the amount of JCL a user must supply himself. If, for example, 'SETL' is the name of a file of JCL statements to invoke the SETL compiler, the data for a SETL compilation could be preceeded by:

{<'label','C'>,<'command','EXEC'>,<'PROC','SETL'>}

PROCs may be thought of as JCL macros. An EXEC statement must have either a PROC subfield or a PROG subfield, but not both.

The PARM Subfield

This optional subfield passes a tuple of parameters to the job step. It is coded:

<'PARM',<a_1,...,a_n>>

The main program of the job step can reference one argument, which will be the parameter PARM.

The COND Subfield

This optional subfield is coded:

<'COND',<'r',n>>

where **r** is an arithmetic relational operator and n is a positive integer. If the immediately preceeding step returned a result x to the system, and the relation x **r** n is true, then this step is executed; Otherwise the job is terminated. If the COND subfield is not given, the default is '**ne** 0'. A sample job step statement using the COND subfield is

{<'label','T'>,<'command','EXEC'>,<'PROG','RUN'>,<'COND',<'le',256>>}

The TIME Subfield

This optional subfield is coded exactly in the same form as the TIME subfield for the JOB statement. If TIME is used for a job step statement, the maximum time allowed for the step is the minimum of the TIME subfield for the step and the total time remaining for the job. If time runs out for the step, the step is terminated, and the job proceeds with the next step. If TIME is not used, the step is allowed the total time remaining for the job.

3.2.3 The Data File Statement

Before giving a detailed description of the data file statement, it is necessary to discuss the various file-name spaces which we suppose to exist in our computing system. At the physical level, names refer to input/output devices attached to the computing system, and to dismountable volumes, such as tape reels or disc packs. To physically address a file on permanently mounted storage, one must specify the hardware address of the device on which the file is stored. To physically address a file on dismountable storage, first the volume on which the file resides must be specified; then an appropriate device on which the volume can be mounted must be specified. Once the mounting has been accomplished, it suffices to address only the device on which the volume has been mounted.

For the user, it is more convenient to associate names with files in a hardware independent and volume independent manner. In the case of dismountable storage, it is not to the user's advantage to permit him to specify on which of several identical devices to mount his volume since this would prevent his job from running when the specified device type is available, but the specific device is not.

The system catalogue provides a map from the user file name space to the physical storage address space. Physical addresses may be of several types, some of which identify the volume on which the file resides, identify the areas on the volume allocated to the file, identify the unit on which the

volume is mounted, and identify the channel through which the unit transmits data to main memory.

Once a physical file has been described to the catalogue via the data file JCL statement, future references need only give the user file name. The operating system, by use of the catalogue, will determine where the file is stored, and in the case of dismountable storage, it will allocate an appropriate device for the file.

When a program is written it is not always clear which files will be used with it. The relation between program file names and user file names is established by the data file JCL statement. At execution time, the correspondence between program file names and physical devices is given by the composition of the system catalogue and the job's data file JCL statements.

The form of the data file statement is:

{<'label',nm>,<'command','FILE'>,a_1,...,a_n}

In the above prototype, nm is a name in program file name space.

The NAME Subfield

Each data file statement has a name subfield which specifies the user name for the file to be associated with the program file name. This subfield may be given in three ways:

1. A symbolic name may be given, e.g.:

{<'label','INPUT'>,<'command','FILE'>,<'NAME','MASTER'>}

In this case, if the parameter in the name field is found in the catalogue, the correspondence with a physical file is determined from the catalogued information. Otherwise a new entry is made in the catalogue, using the additional subfields on the statment to define the physical file. If no additional subfields are given, the operating system will assign the physical space for the file. For a scratch file, the name parameter may be omitted.

2. A reference to a previous FILE JCL statement, usually in a previous job step, may be given. In this case, the information from the previous JCL statement is used. This is specified by giving the step name or the proc name, followed by a period, followed by the program name used in that step. For example, suppose that all language processors store the machine language output in a program named file called TEXT. The loader step would need to refer to this file also. If the language processor's PROC name were TRANS, then the input file for the loader can be specified by:

{<'label','INPUT'>,<'command','FILE'>,<'NAME','TRANS.TEXT'>}

This manner of describing a file is especially useful when constructing a JCL procedure; the program file names are known in this case, but not the user file names.

3. The data is included in the job being submitted. In this case, an asterisk is used as the user file name, and the data follows the defining FILE statement, e.g.

{<'label','INPUT'>,<'command','FILE'>,<'NAME','*'>}

See 3.2.4 for the end-of-data specification in this case.

While a user u may give a file the name x, the file is known within the system as <u,x>. In this way, two users inventing the same file name will not inadvertantly share the same data. (Of course, there may be instances when several users wish to share data. Additional command language to specify such actions are discussed in section 5.4.)

The DEVICE and VOLUME Subfields

The DEVICE subfield specifies the device **type** to be used for the file, but **not** the physical device address. In our system, we will allow the following types:

CARDIN for card readers,
CARDOUT for card punches,
PRINTER for high speed printers,
TAPE for tape drives, assuming only one type of tape drive is available,
DISC for disc drives with removable disc packs, assuming that only one type of removable storage disc drive is available, and
PERM for permanently mounted file storage.

The VOLUME subfield gives a physical identification of the volume on which the file resides, and identifies the volume to the computer operator. For example, if a user wishes to establish a new file on tape reel 2048, he might specify:

{<'label', 'OUTPUT'>, <'command', 'FILE'>, <'name','RESULT'>,
<'device','TAPE', <'volume','2048'>}

In the above example, if VOLUME were not specified, the system would assign an available tape reel.

The DISPOSITION Subfield

This subfield is used to specify whether or not the file is to be 'rewound' before the step starts. It also specifies whether or not to catalogue the FILE statement, that is, whether or not the file is to be retained by the system after the job step is completed. If the system retains the file, the information which relates physical storage to the file is kept in the catalogue, and in subsequent runs, the user need give only the file name in JCL statements in order to describe the file to the system. 'NOCAT' causes the system to purge information concerning the file from its catalogue. If the file had been stored on a shared volume, the file is lost after the job step is terminated. The possible contents of the subfield are:

LEAVE don't rewind

REWIND rewind before use - the default value
CAT catalogue - the default value
NOCAT don't catalogue, or drop from catalogue

If more than one of the above are to be given, they should be the components of a tuple, e.g.

{<'label','TEXT'>,<'command','FILE'>,<'name','TXTFILE'>,
<'device','TAPE'>,<'disp',<'LEAVE','NOCAT'>>}

The SPACE Subfield

This subfield is coded

<'SPACE',n>

where n is an integer which indicates the amount of space to be allocated to the file. If this allocation proves to be too small during execution, the system will attempt to allocate additional space on the same volume. ALL may be given instead of n to indicate allocation of an entire volume. This is taken as the default for tape.

3.2.4 End-of-File Statement

This statement is coded:

{<'command','end-of-data'>}

After encountering a JCL file statement with the name subfield <'name','*'>, all information read is regarded as part of that file until the end-of-file statement is encountered. The end-of-file statement is also used to terminate a catalogued JCL procedure.

3.3 Monitor Services

Services which the operating system can perform on behalf of the user are invoked by statements of the form:

monitor(service,arguments);

where 'service' indicates the action to be performed and 'arguments' are additional parameters describing the service desired. Rather than assuming that such a request invokes a subprogram in the conventional manner, we will assume that such a statement causes a special interrupt, and that the list of parameters gets passed to the appropriate interrupt handler through **cause**.

For several reasons we prefer to use monitor-interrupt linkage, rather than the subroutine call, when monitor services are requested by users. Foremost, by using interrupts, no portion of the operating system need be directly addressable by a user program, as would be the case if subroutine linkage were used. This allows the internal structure of the operating system and the names of internal subroutines to remain invisible to the user, but he can request that the operating system perform actions on his behalf.

The only services which we will describe explicitly in this system are step termination requests, and input-output requests. The table below lists the services supported.

Step termination services (Section 3.3.1)	
endstep	abend
Input-output services (Section 3.3.2)	
read	enable
write	disable
backspace	wait
rewind	iointerrupt
space	fixup
release	endfixup

3.3.1 Step Termination

Normal step termination is invoked by:

```
monitor('endstep');
```

and abnormal termination by

```
monitor('abend',x);
```

where x is a numeric code which indicates the reason for the termination. x is also used as a parameter by the job control interpreter in determining whether or not to skip the next job step. In making this decision after a normal termination, a normal termination will be taken to be equivalent to monitor('abend',0);. The code x is returned to the job control interpreter via its workqueue.

3.3.2. Input/Output

Most operating system environments provide a variety of I/O facilities, including facilites allowing the user to specify how computing and I/O are to be overlapped. In some multiprogramming environments, the ability for a user to specify CPU-I/O overlap in such detail is not important, since global efficiencies can be realized by running CPU and channels concurrently, but not necessarily all for the same process. In our uniprogramming environment user controlled I/O overlap is essential, since it represents the **only** opportunity to use parallelism within a user program. Hence our basic I/O facilities must support detailed control of CPU-channel overlap. A user interrupt facility therefore is an integral part of the supervisor services which are provided to the user; this facility corresponds to the privileged PSETL interrupt mechanism which is described in section 2.2.2.

Data for I/O operations, such as the string to be written on a file as the result of a write-request, or the string to be read from a file as the result of a read-request, are transmitted between the operating system and the requesting process via the requesting process's workqueue. I/O related items on the workqueue will be of the form <b,data>, where b is a blank atom, and 'data' is the information being transmitted. A function, **buffer**, given below, shows how items are fetched and stored on the workqueue.

```
definef buffer b;
(load) /*Fetches the item paired with b on this process's workqueue. If <b,data> is on the
       workqueue, then buffer b returns data.*/
    findfirst(thisprocess, x, x(1) eq b);
    if x eq Ω then
        return Ω;
    else
        remove(thisprocess, x);
        return x(2);
    end if x;
end load;

(store result) /*Forms the pair <b, result>, and places it on the process's workqueue, making
       sure it is the only tuple t on the workqueue such that hd t eq b.*/
L:  findfirst(thisprocess, x, x(1) eq b);
    if x ne Ω then
        remove(thisprocess, x);
        go to L; /*to look for other items x with x(1) eq b*/
    else
        putlast(thisprocess, <b,result>);
        return;
    end if;
end store;
end buffer;
```

where 'thisprocess' is given by the macro:

```
macro thisprocess; CPUcontrol endm thisprocess;
```

The monitor call:

```
monitor('read', f, b);
```

causes the next record of file f to be read and associated with the blank atom b on the calling process's workqueue. The read monitor request only **starts** the I/O operation. The user can synchronize his program in several ways with the termination of the operation, at which time the string of characters associated with b represents the record just read. A monitor wait service is provided for this purpose. For example:

```
    .
    .
monitor ('read', f, b);
    .
    .   /*computing, but not on buffer b.*/
    .
monitor('wait', f); /*Delay until operation on f is completed.*/
r=buffer b; /*get the info just read.*/
    .
    .
```

The wait service is described more fully below. A second method involves a user's interrupt facility, which is also described below, and which is illustrated by the output spool code in Section 3.5.3.

To write a record to file f, a monitor call of the form:

```
monitor('write', f, b);
```

is used. The string of characters, s, associated with the atom b on the calling process's workqueue is written on file f as the next record. An assignment statement such as:

```
buffer b = s;
```

may be used to place the string s on the workqueue prior to the 'write'. As with the read request, write requests only start the operation. One of the synchronization techniques mentioned above must also be used.

The monitor request:

```
monitor('backspace', f);
```

causes the file to be repositioned to the previous record, i.e., if f contains records r_1, r_2, ..., r_n, and if f is positioned at r_j, the result of a backspace will leave f positioned at record $r_{1 \max j-1}$.

The monitor request:

```
monitor ('rewind', f);
```

repositions f to the first record of the file (record r_1 in terms of the above example). For files which are on devices such as card readers or printers, these commands are inappropriate, and cause an error to be indicated when the operation terminates.

The monitor request:

```
monitor('space',f);
```

positions file f to the beginning of a new page, when f is on a printer or similar device. If this request is issued for a device which is not a printer or something similar, an iointerrupt code (see below) indicating an illegal I/O request will be returned.

The user I/O interrupt system is normally enabled. Two monitor call statements are provided with our operating system to alter the state of the user I/O interrupt system for each program file:

```
monitor('enable', pfn);
monitor('disable', pfn);
```

While a file is disabled, interrupts resulting from the termination of an I/O request on pfn are stacked by the operating system, to be released FIFO when the interrupt system for pfn is re-enabled by the user.

In some cases it may be desirable to ignore the interrupt information associated with an I/O operation. (For example, when rewinding a file, the interrupt bits are generally uninteresting.) In particular, when a file is disabled, or while executing a fixup routine (see below), a mechanism to avoid stacking certain interrupt reports is needed. The 'release' monitor service call,

```
monitor('release', pfn);
```

causes stacked interrupt information for pfn to be discarded, or, if pfn is currently active, it will cause the interrupt information resulting from the current operation to be discarded.

If a user has specified an I/O interrupt routine for a file by having executed:

```
monitor('fixup',f,r);
```

control will be forced to the statement whose label is r on the occurrence of an I/O interrupt due to the file f, whenever a monitor(disable) is not in effect. At r, the user program can get the cause of the interruption via the call,

```
monitor('iointerrupt',y);
```

which results in **buffer** y being set to information concerning the cause of the most recently processed user I/O interrupt (interrupts stacked are not considered in determining the most recently processed interrupt). **buffer** y will be a pair, whose first component is the program file identifier for the file which caused the I/O interrupt, and whose second component is machine dependent and gives information regarding the completed I/O operation. Since monitor('iointerrupt',y) will set **buffer** y to the information related to the most recent file to have caused entry to a user fixup routine or to have satisfied a wait-request, this monitor call should be one of the first actions taken in an I/O interrupt routine. We will use the term 'fixup' to designate code executed as the result of interruption at the termination of an I/O operation on a file. If no fixup has been specified for a file by a user, then a standard system fixup is used.

An I/O fixup routine may be regarded as a co-routine of the mainstream program. All of the variables of the mainstream program are available to the fixup and vice-versa. When control enters an I/O fixup, our monitor will save the mainstream location counter, and restore it when the user program executes:

```
monitor('endfixup');
```

While in a fixup routine, further user I/O interrupts are inhibited and any interrupts which occur during fixup are stacked by our operating system, to be released FIFO on execution of monitor('endfixup');.

To synchronize CPU and I/O activities,

```
monitor('wait',f);
```

is used to suspend the calling process until the file f is released by the system. If f was not busy when the 'wait' was executed, the 'wait' behaves like a no-operation. Otherwise, control goes to f's fixup when the operation on f is concluded. If an interrupt is already stacked for f, it is unstacked and control reaches f's fixup, regardless of the state of the interrupt system. By disabling interrupts and issuing 'wait's, a user program can handle interrupts in a predictable order. Note that waiting on a file f causes the disabled condition to be overridden for file f only. If a 'wait' is issued while in a fixup routine, and the wait is not a no-op, then the process resumes in the new fixup, and the old fixup is considered to be concluded.

Finally, a program file f is considered to be busy from the point at which an I/O operation is accepted to the point where its fixup routine is entered. Thus, it is allowed to issue an I/O instruction to a file as part of its fixup routine.

3.4 System Organization

The structure of our operating system can be likened to a set of concentric circles. The innermost circle represents programs which have the highest privilege level and which are the most machine dependent. Each outer circle has a lower privilege level than any circle it contains, and tends to be more machine independent, with operating system programs in the outermost shell being indistinguishable from user programs.

3.4.1 System Nucleus

The innermost circle, or system nucleus, consists of the basic interrupt handling programs, protection mechanisms, functions and maps describing the structure and status of the hardware being controlled, and a rudimentary dispatcher.

The interrupt handling processes are precisely those specified in **hd tl[interrupt]**. In most cases, interrupts are handled by posting information on the workqueue of a process operating in an outer circle.

Data sets associated with the nucleus describe the hardware to be controlled. Thus, if c a channel, devices{c} might give us the hardware addresses of all devices on channel c, etc. Using these maps, we can express many of the common operating system operations without becoming enmeshed in undue detail involving the precise layout of the tables in storage.

3.4.2 Resource Allocation

The next highest privilege level of our operating system handles resource allocation. To this level belong our main library catalogue and routines to search or update this catalogue.

Resource allocation routines accept the description of a device type along with certain restrictions, and return the physical address of a device which satisfies the description. Given the user defined name of a file, the catalogue identifies the physical volume or device which contains the file.

The data and routines at this level are available to processes operating in the next lower privilege level, but not to processes with still lower privilege. Such processes achieve resource allocation by asking a higher level process to request the allocation for them; the higher level process can then perform consistency checks and impose address mappings before actually requesting allocation.

3.4.3 Major Components

The next highest privilege level of our operating system consists of the major system components. Our operating system allows several major activities to take place concurrently with the running of a user program. These are: communication between machine operator and system, reading of input for subsequent jobs, printing of output from previously run jobs, allocation of resources, and scheduling. Corresponding to each of these activities there will exist a program and one or more

processes. For example, the program to read input may be executed by as many processes as there are card readers.

3.4.3.1 Operator Communication

Even though operating systems are produced to automate the running of a computer installation, some communication between operator and operating system will be necessary. Operator communications will fall into several general catagories: communications to the operating system giving information about the physical environment, requests overriding the operating system's automatic operation, and inquiries to the operating system on the status of jobs or workqueues.

The operator is expected to inform the operating system whenever he has mounted a volume on a device. To initiate the running of jobs, he must indicate which of the devices are to be the system input and/or output devices.

In our system, it will also be the operator's responsibility to communicate to the system the eligible users, along with profile information for each user. This information includes the maximum priority level at which the user may run, as well as information concerning user-chosen defaults to be used in conjuction with JCL statements. The operator also has a command whereby he can allocate time to each user. A user whose jobs have used up this allocation cannot execute additional jobs until the operator issues him a new allocation of time. The operator can advance a user's job to the top of the scheduler's highest priority queue. Finally, the operator has a command to remove a user from the set of eligible system users. This command results in purging all system tables of data which pertains to the former user.

The set of operator commands to which our system responds is by no means complete, but serves to illustrate interaction between the operator and the system. Additional functions are discussed in Section 4.6.

3.4.3.2 Scheduler

In our simple uniprogrammed environment, we use a relatively straightfoward scheduler. When the CPU finishes a job, the scheduler selects the oldest member from the set of highest priority jobs. If mounting of volumes is required, messages are sent to the operator, and an estimate of the time required to mount the volumes is computed. The scheduler then searches for a job whose estimated running time is less than the mounting time and which does not require operator setup. If such a job is found, it is run during what would otherwise be idle CPU job transition time.

Our scheduler also uses the catalogue and JCL to match names in program file name space with those in physical address name space. When allocation for new data files is not specified, the scheduler will make an appropriate assignment. If the computing system has inadequate resources for a job, the scheduler will cause the job to be terminated.

3.4.3.3 Input Reader

To reduce job transition time and job execution time, our operating system will read jobs being submitted through input devices and copy the jobs onto disc storage, from which the data can be read faster than from the original input device. During the transfer of data from input device to disc, the input reader will analyze all the JCL statements it finds, and will determine the resources

which the job will require; it will then summarize this information for use by the scheduler. If JCL procedures are used, the input reader will locate the procedures in the library and expand the procedure references into standard JCL. On detecting the end of a job, the scheduler will be notified through its workqueue that another job is available for scheduling consideration. The entire job, considered as a data file, will at the same time be catalogued for later reference by those of the program's read statements which are directed to the data included in the job.

This mode of buffering jobs on intermediate disc storage assumes two things: that disc storage can be read substantially faster than the system input devices (which are often card readers or slow communication lines), and that jobs are likely to be presented while a backlog of work exists for the CPU. The reading of a job on a slow device during a time when there is backlog can be overlapped with the execution of the backlog. Furthermore, the greater the backlog, the greater the selection of work for the scheduler in its attempt to utilize hardware effectively.

3.4.3.4 Output Printer

If we assume that because of the relatively slow speed of printers, the time to print the output for a job tends to be longer than the CPU time needed to execute the job, then we can expect to increase the performance of our system by writing information intended for the printer onto a fast I/O device, such as a disc, and by copying the disc file to the printer while subsequent jobs are executing. Several printers may be required on a computing system with a very fast, powerful CPU. Our operating system will incorporate such a buffering scheme for printed output.

Whenever a job finishes executing, the output printing routine will be alerted. If a printer is available, the output is sent to it; otherwise the request for printing gets stacked. When the printing of an output file is terminated, the output printer will unstack the request for the highest priority job; if several stacked requests for the same priority class are present, the oldest will be chosen, that is, FIFO queue discipline will be maintained within each priority class.

3.4.4 User Programs

User programs operate at the lowest privilege level. Some software which the operating system provides also operates at this level; examples are language processors, sorting subsystems, and some portions of data management systems. These portions of operating systems will not be depicted at this time.

3.5 Coded Operating System

3.5.1 Sets, Maps, and Tables

In this section, we describe the sets, maps and tables which our operating system uses to describe the hardware configuration on which it runs, the states of the various devices which it manages, and the jobs which it controls. These structures will appear in the code which follows, and will serve as a major communication channel between the various components of our system. Of course, the special sets described in Chapter II will also be present.

3.5.1.1 Global Information

users: The set **users** contains the identifiers of all bona-fide users of the computing system. Each element of this set is a character string.

owner: This set consists of pairs of the form <m,u>, where m∈**movers** and u∈users. This map is many:one since one user may have several jobs in the system.

devices: This set contains the hardware addresses of all I/O devices which are attached to the computing system.

channels: This set contains the hardware addresses of all data channels available to the computing system. (We assume that a channel is necessary and sufficient to establish a data path for transmission of data between a device and main memory. We allow several devices to be attached to the computer through the same channel, although the channel can act as a communication path for only one device at a time.)

type: This set is used to classify devices. For d∈devices, type(d) identifies the nature of the device. In our system, we assume the following device types: card reader, card punch, printer, tape drive, disc (for disc drives which use dismountable disc packs), and perm (for disc drives which have a fixed storage surface).

volumes: This set contains one element for each storage element available to the system. Each element is represented by a tuple <id,t,s> where 'id' is the volume's external identification, t is the type of device on which it is used, and s is the amount of unassigned space on that volume.

3.5.1.2 User Oriented Information

budget: For each u∈users, budget(u) is the remaining time available for jobs submitted by u.

defaults: For each u∈users, defaults{u} is a set of pairs, giving the user's specified default values for various JCL parameters.

datafiles: For each u∈users, datafiles{u} is the set of file names of data files accessible to the user u.

catalogue: Members of the set 'catalogue' are of the form <d,v,k>, where d∈datafiles, v∈volumes, and k is a set of pairs, each pair consisting of an attribute name and a value associated with that attribute. Examples of attribute names are: extents, private volume, etc. The value associated with 'extents' is a list of pairs giving the location and size of the physical storage blocks for the file. In the event that non-contiguous blocks are used to contain the file, the order of the pairs determines the logical ordering of the file.

maxprio: For u∈users, maxprio(u) gives the maximum priority level which the user u can assign to his own jobs.

3.5.1.3 Process and Mover Oriented Information

userprogs: This set contains those process identifiers which correspond to active user programs.

esttime: For each mover m, esttime(m) gives the estimated total CPU time for the mover as given by the job statement.

timeleft: For each mover m, timeleft(m) is the CPU time remaining for m.

steptimeleft: For each process p, steptimeleft(p) gives the CPU time remaining for the job step associated with p.

iowait: For each process p, iowait(p) is **true** if and only if p is suspended awaiting completion of an

I/O event.

mainstate: For a process p, mainstate(p) is the state of process p's mainstream execution. If p handles its own interrupts, then mainstate(p) is used to return to the main processing program when the interrupt fixup routine is completed.

ioint: For a process p, ioint(p) is **true** if and only if p's location counter points into an I/O interrupt fixup routine. In such case, other I/O interrupts for process p will automatically be stacked, and handled FIFO whenever the process p attempts to return from an interrupt handler to the program's 'mainstream'.

workset(<p,io>): For a process p, this workqueue is the queue of stacked I/O interrupts.

programfiles: For process p, programfiles{p} are the names in program file name space being used by p.

newallocations: For each mover m, newallocations(m) is the set of user files which are being catalogued for the first time in the job associated with m. If any of these files are never used, they are automatically removed from the catalogue when the job corresponding to m finishes.

interrupted: For process p, interrupted(p) is a pair of the form <f,i>, where f is the most recent file for which an I/O fixup routine was entered, and i is the interrupt information associated with file f.

3.5.1.4 Channel and Device Oriented Sets

operational: For x∈{devices + channels}, operational(x) is true if and only if x is usable by the hardware, i.e. "x is up".

units: For c∈channels, units{c} is the set of addresses of devices on channel c. These addresses are hardware imposed names of the devices, and are used to reference the devices in physical I/O statements (see section 2.3.2).

filehandledby: For d∈devices, filehandledby(d) identifies the member of programfiles on whose behalf d is being used.

position: This is a device dependent set which for a device d returns the position at which the next operation will take place. This set is used primarily for devices {d∈devices | devicetype(d) **eq** tempdisc}.

rightouse: For d∈devices, rightouse(d) is an atom used by the operating system for controlling macro operations on d. Since macro operations may consist of several suboperations, it is the atom which becomes free as the direct result of an I/O interrupt. The operating system will free a physical device d only when a full monitor request on d is completed.

mounted: For d∈devices, mounted(d) identifies the volume logically mounted on device d, unless there is no volume on d, in which case mounted(d)=Ω.

ready: This is the set of devices on which an assigned volume has been mounted by the operator. A device d can be used for I/O if and only if d∈ready.

3.5.1.5 Maps with Domain programfiles

deviceaddress: This set provides a map from program file name space to device address space.

userfile: This set provides a map from program file name space to user file name space.

savecause: For each r∈programfiles, savecause(r) is the information concerning the most recent interrupt caused by file r.

filewait: For r∈programfiles, filewait(r) is **true** if and only if the process to which r belongs is

suspended waiting for an interrupt signal announcing the completion of an I/O operation performed on file r.

fixup: This set provides a map between program file names and user-level I/O interrupt handling routines.

logicalposition: For r∈programfiles, if r is stored on disc, logicalposition(r) describes the logical position within the file at which the user's program assumes that the read/write head is located. The physical position of the read/write head can then be computed by using this information in combination with catalogue(userfile(r)).

rel: For r∈programfiles, rel(r) is **true** if a monitor release is in effect for r (cf. 3.3.2).

disable: For r∈programfiles, disable(r) is **true** if the fixup for file r is not to be automatically executed at the termination of I/O operations on file r.

3.5.2 Remarks on the Uniprogrammed System

Our operating system is organized as a sequence of subprograms. The first subprograms presented are the so called major components. The coding style used in these portions of the operating system closely resemble conventional SETL programs, and do not employ PSETL extensions extensively. As one progresses along the sequence of subprograms, PSETL notions play an increasingly important role, and the reader must keep in mind that several processes are "in execution".

A major difference between these subroutines and many of the SETL algorithms published to date is the relative absence of iteration headers. For most paths through a portion of the operating system, there is simply a **sequence** of computations to be performed, and iterations play a minor role; when they do occur, they have short scope and modest range. On reflection, this is desirable in an operating system, especially the nucleus, if the operating system overhead is to remain small. A call for operating system service should only involve the execution of a small number of machine instructions. The most often traversed portion of the operating system is its interrupt handling section. The fact that our algorithms for interrupt handling are relatively free of iteration gives hope that these efficiency-crucial sections of code can have efficient realizations.

Succinct, comprehensible expression of the data-processing specifications in our system is attained with SETL. Searches over the system's structures are easily expressed in terms of ∃-conditionals. Thus, there is no need, from a specification point of view, to maintain maps whose sole purpose is to invert other maps.

In general, we have tried to avoid redundant structures in our system. For example, at various points, we need a map from program file names to devices. Such a map could easily have been constructed in the subprogram 'assign'. However, we compute the map whenever needed in terms of other structures as follows:

```
deviceaddress(pfn) = catalogue(userfile(pfn))('deviceaddr');
```

The advantage of this construction is to simplify the maintenance of the various system structures. In the above example, we would have had to consider modifying the additional map whenever 'catalogue' or 'userfile' changed, in order to keep the maps consistant.

SETL's strict adherence to reference-by-value does pose serious problems for operating system specification. Some objects, such as process environments and files, which must be manipulated, can be very large. For example, since file variables include all the data in a file, rather than a pointer to the data, it becomes impractical to construct a list of files, since this implies generating copies of all the data in the files. Input spool does require such a construction, and it resorts to using a list of program file names, together with provisions for re-opening and repositioning files. These representation-dependent computations are not in keeping with the spirit of SETL, but they are unavoidable if we don't want to specify an unrealistic copying of files.

The PSETL definition of disabled blocks removes any doubt whether the system will leave the disabled state, at the expense of greatly restricting the dictions which can be used in a disabled block. The lack of subroutine calls is particularly frustrating, and leads to repetitive code, as for example, the time accounting at the beginning of the various interrupt handlers. The lack of while-blocks sometimes requires ∀-iteration headers which specify an overestimate of the number of iterations of the block.

3.5.2.1 Input Reader

The subprogram "inputspool" is initiated once for each device which the operator designates as a system input device. Thus as many processes will execute "inputspool" as there are system input devices. Each "inputspool" process is passed a user file name which has been initialised to address the physical input device which that process is to use. The private variable "reader" is used as the program file name corresponding to the input device. Whenever a new job is encountered, a new mover is created. A user file is created to hold the job-associated JCL.

By reading jobs from external, non-random access sources, such as cards, and storing the jobs on disc storage, the input processes create a reservoir of work for the scheduler to choose from. This makes jobs to be run available in a manner implying the absence of any essential time difference in selecting one job over another, and on a medium from which input records can be read at speeds better matched to processor speeds than a card reader would allow. At the same time, the input readers extract certain key job characteristics, such as estimated running times and the names of the required files, and keep these in main memory. As input is stored on temporary disc, certain modifications to the source material are made in it. We allow JCL macros, or 'proc's (see 3.2.2); if such a macro occurs, it is expanded by the input reader, which copies the necessary macro expansion from the user's or from a system library. Data included in a "job deck" (such as a source program to be compiled) are written to separate data files, and not included in the disc storage copy of the job. Hence the disc copy of the job contains only JCL statements.

In order to guarantee that job steps have unique names and still allow a single macro to be invoked several times within a job, the following technique is used. During JCL expansion, step names are prefixed by x+'.' where x represents the label on the input JCL 'exec' statement which initiated macro expansion. Thus, the use of unique labels in the source deck is sufficient to guarantee unique step names after macro expansion. The variable 'qualifierprefix' holds the string to be prefixed to JCL labels in accordance with this scheme.

Since input can come from either the system input device or from a JCL macro library, the tuple 'source' is used as a stack of input sources, with source(1) being the current input device. The predicate #source **eq** 1 (or equivalently, the macro, readingprimarysource) is used to distinguish

whether input is being read from the system reader or a JCL macro library.

The main loop of the input reader starts at 'mainloop'. At this point a JCL statement is read and copied onto a temporary file, following which an appropriate section of code depending on the JCL command encountered is executed.

Control reaches the label 'endcard' when a JCL 'end' statement, or the header statement of a second job in encountered. (A header statement not preceeded by an end card serves the dual purpose of signalling the end of the present job and the beginning of the next.) If macro expansion had been in progress when the end card was read, the previous input source again becomes the current input source, and the main loop continues.

If a JCL error has been encountered during reading of the job, the job is rejected out of hand. In this case, the temporary input file, which now also contains error indications, is sent to 'outspool' for printing, and all table entries and temporary files for the rejected file are purged.

'exec' statements are checked for the 'proc' parameter. If present, then the user's library as well as the system library are searched for the JCL macro. The user's library is searched first, so that a user written JCL macro will take precedence over one with the same name supplied by the system. It is assumed that the procedure library, both for the user and the system, is named 'proclib'.

In handling file statements, various steps may utilize the same program file names; JCL macros, consisting of several steps whose details are not known to the user, create this situation. To guarantee unique program file names, the program file name on the file statement is prefixed with the step identifier. The processing of file statements falls into two main classes.

If the user file name contains a '.', then a reference is being made to a previously encountered file statement, and a check is made to insure that the referenced file statement had indeed been encountered (see 3.2.3).

If the user file name is given as '*' or if it is omitted (for a scratch file), a userfile name is generated for the file. If the user file name is found in the catalogue, then items not specified on the file statement are given default values based on catalogue entries. Otherwise, a new catalogue entry is made, and parameters not specified in the file statement are defaulted from the user's profile.

Finally, if the user file name had been given as '*', space is assigned for a new file, and the input is copied into the new file, until an end statement is encountered.

3.5.2.2 Output Writer

The subprogram 'outspool' is initiated once for each device which has been designated as a system output device. To illustrate the direct use of monitor supplied I/O functions and interrupt facilities, this subprogram is written using the primitive monitor calls described in 3.4.1.7, rather that the common SETL I/O statements. (The usual SETL I/O really invokes subroutines which must be actually coded in much the same style as the output writer.) Given a user file, 'ufn', to be printed, a program file name, 'pfn', is assigned to the file. The bulk of processing takes place in the fixup routines which receive control on completion of I/O operations. Control returns to mainstream either when the end of file for the input is reached ('eof' is **true**) or during a read error recovery attempt ('eof' is **false**), in which case mainstream merely re-enters the wait condition.

On disc I/O interrupt, if the record has been successfully read, another read is immediately initiated into the other buffer (determined by 'useleftbuffer'), and a wait is entered for the printer. Thus, control reaches 'printerfixup' without ever returning to mainstream. The appropriate buffer is printed, and the program waits for the reading of the next disc record to be completed.

3.5.2.3 Scheduler

Only one instance of the scheduler exists (as opposed to the multiple instances of input readers and output writers). Items are placed on the scheduler's workqueue when an input reader finishes reading a job which is JCL error free, when a non-setup job is needed to overlap mounting time for an already scheduled job, and when a job terminates.

New jobs from an input reader are kept in 'backlog' which is ordered according to job priority, and within each priority class, according to age, so that the first item in 'backlog' is the oldest job having the highest priority, and the last item is the youngest job having the lowest priority. Each item in 'backlog' specifies the mover for the new job, and its temporary input file.

In response to a request for a non-setup job, the first such job in 'backlog' whose running time is less than the estimated mounting time, if one exists, is selected and is fowarded to the process 'jobcontrol' for execution.

For other requests, if no other user program is running, the first item of 'backlog' is selected. Provided that a sufficient number of devices for the job are present on the system, the job and its input file are fowarded to a newly created process for the new job's mover, called 'jobcontrol'.

3.5.2.4 Job Control

Job control reads the JCL statements for one jobstep of a job, sets up the files and loads the required programs. Job control is activated as a process belonging to the mover for which the next jobstep is to be run. The subprogram is passed the user's jcl file name and output file name as parameters. There is structural similarity to the input reader, since JCL statements are being read and interpreted, but procedure libraries are no longer involved since the input reader has already expanded all JCL macros.

For all file statements, if the reference is to a previously defined program file, the file description is obtained through the map 'userfile' and the catalogue. If the file is already mounted, nothing further need be done. Otherwise an appropriate available device is selected, a mounting message is sent to the operator, and one minute is added to the estimated mounting time for the jobstep.

The end of step is signalled by an end statement, or by the next 'exec' statement. If there is going to be delay due to device mounting, the scheduler is invoked to select a non-setup job, and it is passed the estimated mounting delay as a parameter.

Finally, loading of the program is performed. This is too strongly dependent on the 'standard' form of compiler output and structure of the hardware to be shown in complete detail. However, the general nature of the loading process is shown from the statement at 'loadloop' to the if statement following the while block. The variable 'progstobeloaded' is initialized to contain the names of the programs to be loaded, and 'missing' initialized to nl. All items in 'progstobeloaded'

are searched for, first in the user's program library, then in the system's. The actual loading of a found program is accomplished by the routine 'fetch', and the machine dependent parts of loading are accomplished therein. We also assume that for any program being loaded, as many inter-program references as possible are resolved. Reference to a program which is not already loaded and not given in 'progstobeloaded' is noted by placing the name of such a program in 'missing'. Inability to find a program in either user or system catalogue also results in the program being noted in 'missing'. So long as only new names enter 'missing', 'loadloop' is repeated. Failure to find a program results in termination of the job step. It is also assumed that 'fetch' initializes the variable 'main' to the entry point of the (composite) loaded program.

The final block of code in job control sets up the time constraints which bound the running of the job step just loaded, and which would force return of control to 'jobcontrol' in the event of a time overrun.

3.5.2.5 Resource Allocation

The four subroutines, assign, unhook, allocate, and relinquish, are involved in maintaining maps between program file name space, user file name space, and physical storage such as disc space and tape reels.

The subprogram 'assign' establishes the correspondence between a program file name and a user file name. Several system structures are updated in establishing the correspondence. The program file name being defined is put into 'programfiles', the set of program file names which are currently associated with user file names. The map 'userfile' is updated to indicate the user file name associated with the program file name.

The program 'unhook' breaks the relation between a program file name and a user file, and also destroys the catalogue entry if the disposition field of the catalogue does not indicate that the file is to be kept in the system.

The subprogram 'allocate' assigns space on a disc pack or a tape reel to a file. If the catalogue entry for the file contains a volume entry, no action is taken. Otherwise, if the device type for the file is disc or drum, an attempt is made to find space on a mounted volume, but if this fails, then space on an unmounted volume is sought. A similar strategy is followed for tape files, except that an entire reel is always allocated.

When a disc file is being released, the subprogram 'relinquish' updates the volume table of contents of the volume on which the file had resided, to indicate where additional available space may now be found. A system structure, 'volumes', is also updated to reflect the additional available space.

3.5.2.6 Operator Services

When the operating system itself begins execution, the operator services process is given control. It's first task, handled in the **initially** block, is to initialize the three interrupt handlers. An initial environment with system privilege, 'osenvironment', is given to these processes.

The main loop begins with a read operation directed to the operator's keyboard. The actions associated with most of the commands is straightforward.

The 'mounted' command, which indicates that the operator has performed a volume-mounting operation, is a cue to the system to check whether the action was in compliance with a previously issued mounting request message. For disc devices, it may be the case that while the volume had been dismounted, files had been assigned to it. In such a case, space allocation for these files can now be completed, and the system table which gives the size of the largest block of contiguous storage on the volume is correctly updated. In any case, the device is marked as being ready, thereby signalling the job control interpreter that a mounting action for which it may be waiting is completed, and informing the scheduler which files are currently on-line.

3.5.2.7 Monitor Services

For each monitor-interrupt, a new process to service that interrupt is created by the monitor interrupt handler, which then sets itself up to receive another monitor-interrupt.

The process which requested the monitor service ('caller'), the requested service ('fcn'), and additional parameters ('parm') are extracted by the new process from information passed to it by the monitor interrupt handler via the **split** statement. The first if-statement separates most I/O requests from the others. An illegal request is treated as an abnormal step termination.

Step terminations are processed beginning with the code at label 'endstepaddr'. The cause of the termination is put on the workqueue of the job control interpreter process which started the terminating step. By not returning the caller to the CPU's workqueue, the caller does not regain control of the CPU. The job control interpreter, on the other hand, resumes operation since it's **await** (see 'End of Step' section of code in the Job Control Interpreter) is now satisfied.

For I/O requests, the program file name, paired with the caller's process identifier, produces a unique identifier, 'file', for the file involved in the operation. 'file' is used to determine the device and channel which are involved in the I/O operation. Certain control requests, wait, release, enable, and disable, do not physically or logically manipulate the device, and these requests are now interpreted without further ado.

For the other I/O requests, the user's program file is first reserved. If the file is busy when the reservation is attempted, this process will be delayed until the previous operation on the file has been completed.

Operations which merely reposition the read-write head of disc devices will not result in physical manipulation of the device, but rather, such operations cause system tables to reflect where the caller believes the read-write head to be located. This strategy is adopted because the disc drives are frequently shared among several files. Moving the read-write head for one file may be futile if it is again moved for another file before advantage is taken of it's having been positioned for the first file. Thus, for disc files, we will carry out the read-write head positioning only if needed, before a read or write operation. For such positioning requests, no further facilities must be reserved.

For all other cases, we reserve the device on which the file resides and the channel through which the device communicates with the CPU and main memory (to prevent concurrent use by competing processes). We also reserve a blank atom, given by 'rightouse(iodevice)' for device 'iodevice'. The need for this facility arises as follows: In response to an interrupt from an I/O device, we

want the I/O interrupt handler to free a facility. If it frees the device itself, then a competing process can begin an operation on the device. But some I/O requests require a sequence of physical I/O actions on a device without intervening actions from other I/O requests. (On discs, for example, it may be necessary to issue a read-write head positioning command (see the if-statement following the label 'readwrite') before performing a read or a write operation.) Thus two facilities are needed for each device: one to be used to reserve the device for a process (the device identifier, 'iodevice' serves this purpose), and another to coordinate a monitor process with its I/O operations ('rightouse(iodevice)' serves this purpose). The I/O interrupt handler issues **free**s to the second of these facilities whenever an I/O interrupt occurs (see the code at label 'ioxpt'). By reserving rightouse(iodevice) a monitor process delays its execution until the current physical I/O operation has finished. When an I/O monitor request has been completed, then the monitor service interpretor process **free**s the other facility, 'iodevice', thereby allowing other processes to access that device (see the code at label 'wrapupiorequest').

3.5.3 The Code

```
/*
```

Catalogue of Routines


```
*/
```

```
scope operatingsystem;
macro establish(ufn,pfn,filevar);
    /*For user file ufn, a program file name pfn is established for the running process, and filevar
    becomes a file variable representing the file*/
    assign(<CPUcontrol,newat> is pfn>,ufn,standardfixup);
    filevar=<open pfn,1>;
    endm establish;

macro processparameter;
/*reference a processes initial argument*/
    info(initialvar(state(CPUcontrol)));
    endm processparameter;

macro filesize;
/*default size of standard system-created files*/
    100
endm filesize;

macro primarysource;
/*for the input reader, reference to main input device*/
    hd(source(#source))
endm primarysource;

macro primaryposition;
/*for input spool position in the primary file next to be read*/
    (source(#source))(2)
endm primaryposition;

macro currentsource;
/*for input spool, the current input program-file name*/
    hd hd source
endm currentsource;

macro currentposition;
/*for input spool, the current position in the input file*/
    (hd source)(2)
endm currentposition;

macro readingprimarysource;
/*true iff current source is the system input device*/
    #source eq 1
endm readingprimarysource;

macro thisprocess;
        /*A more mnemonic way for a process to refer to its own process identifier.*/
    CPUcontrol
endm thisprocess;

macro deviceaddress(pfn); /*device on which program file resides*/
    catalogue(userfile(pfn))('deviceaddress')
endm deviceaddress;

/*The global variables declard below are described in section 3.5.1*/
global iowait,mainstate,busystatus,savecause,filewait,fixup
    logicalposition,channels,units;
global 2 volumes, programfiles;
global 3 devices,type,userprogs,steptimeleft,ready,mounted,
```

```
    deviceaddress,symbolicfile,operational,quickrun;
global 4 users, owners, defaults, catalogue, maxprio, esttime,
    timeleft,iofixup,workset,userfile,newallocation,budget;

                /*  ************************  */
                /*          INPUT SPOOL       */
                /*  ************************  */

scope 4 spooling;
    /*This scope contains the procedure which handles input devices such as card readers. Input
    spool has two major functions:
        1. To expand any JCL macros included in jobs.
        2. To verify that the JCL is correct. Errors include illegal
        user name, illegal commands, undefined JCL procedures, and
        duplicated labels.
    For a job being read by input spool, there are two possible outcomes:
        1. One or more JCL errors are discovered. The JCL along with
        diagnostics is printed, and the job is otherwise discarded.
        2. There are no JCL errors. Entries are made in the system
        tables for the job, which becomes enqueued on the scheduler.*/

        /*Each process which reads input is passed a unique data file name which already corre-
        sponds to a physical device acting as a system input reader*/
spoolin: device=processparameter;
    catalogue(userfile(pfn))('deviceaddress')
    assign(<thisprocess,'reader'> is inputfilename,device,standardfixup);
    source=<<inputfilename,1>>;
        /*'source' is a vector of program file names which is used during macro expansion as a
        pushdown list of names of open files, and infile is always the current file*/
    infile=<open currentsource,1>;
        /*Scan ahead to the next 'job' statement.*/
discard:
    (while image('command') ne 'job') infile read image;
        if image eq Ω then
                /*End-of-file while looking for the next job. This will be interpreted as ending the
                spooling operation on this device, and the process terminates. If the operator
                wishes to resume spooling on this device, he may issue a command to start a new
                spooling process (see operator commands).*/
            term;
        end if;
    end while;
newname: /*Check for legal user name.*/
    if n (image('name') is username ∈ users) then
        message( '***'+ image+' reject:illegal user name');
        image=nl;
        go to discard;
    end if;
        /*'job' will identify the new job's mover. Note that we can't use the user's name for this
        purpose, since he may have several jobs in the batch.*/
    job=newat;
        /* initialize
            stepnames (vector of all the job's step names)
            volumesneeded (set of volumes used by this job)
            qualifierprefix (prefix to guarantee unique program file names)
            localfilemap (map from program file names to user file names)
            newfiles (set of files for which new catalogue entry is created).*/
    stepnames=nult;
    volumesneeded=nl;
    qualifierprefix=nulc;
```

```
    localfilemap=nl;
    newfiles=nl;
        /*Initialize error flag to show no errors found yet.*/
    jclerror=false;
        /*Remember that the job card hasn't yet been processed. If another is encountered later,
        it acts as the terminator of the current job.*/
    jobcardprocessed=false;/*signal jobcard for new job*/
        /*Set up a temporary file for job's JCL, in which macros are expanded.*/
    catalogue(<username,newat> is expandedjclcatname)=
        {<'device',tempdisc>,<'space',filesize>,<'disp',{'cat','leave'}>};
    tempfiles={expandedjclcatname};
    allocate(expandedjclcatname); /*file to hold job*/
    establish(expandedjclcatname,'jclopenclosename',expandedjcl);
    go to mainwrite;

                /*  ************************  */
                /*       Main Loop of Input Spool Process      */
                /*  ************************  */

        /* An input statement is read and copied to the expanced JCL file. The command is then
        used to index a branch table, which determines where the statement is further processed.*/

mainloop:  infile read image; /*read input record*/
    if image eq Ω then
        /*end-of-file -- treat as an end-statement*/
        go to endcard;
    end if;
mainwrite:  expandedjcl write image;
    if {<'end',endcard>,  /*This set is a branch table*/
        <'job',jobcard>,  /*for command interpretation.*/
        <'exec',exec>,
        <'file',filecard>}(image('command')) is loc ne Ω then go to loc;
    else
        /* illegal control card -- flag and signal error*/
        jclerror=true;
        expandedjcl write '***illegal command';
        go to mainloop;
    end if;
endcard:
        /*An end statement in a file other than the main input file means end of macro. The
        current program file is closed and the previous one popped up.*/
    if n readingprimarysource then
            /*end macro expansion*/
        unhook(currentsource);
        source=tl source;
        infile=<open currentsource,currentposition>;
            /*remove stepname of the completed macro from prefix*/
        qualifierprefix=if ∃c(i) ∈ qualifierprefix | c eq '.' then
                            qualifierprefix(i+1:) else nulc;
        go to mainloop;
    end if;
        /*end-of-file or end statement while reading main input source, indicating end of job
        found.  If JCL error had been previously discovered, the job is cancelled.*/
    if jclerror then
        close(expandedjcl,'jclopenclosename');
            /*Print JCL and error indications*/
        enqueue expandedjclcatname on outspool for username;
        unhook ('jclopenclosename');
```

```
            /*Discard the temporary data files which had been submitted with the job unless
              disposition was given as KEEP.*/
        (∀y∈tempfiles) unhook (y);;
            /*Discard newly catalogued, but unused files.*/
        (∀y∈newfiles)
            relinquish(y);
            catalogue(y)=Ω;
        end ∀y;
        go to discard;
    end if;
        /*No JCL errors found with job just read. Update system tables from the saved job
        card.*/
    job in movers;
    owner(job)=username; /*Setup map showing job's owner*/
    esttime(job)=tempjobcard('time');
    priority(job)=tempjobcard('priority') min (maxprio(username));
    resources(job)=volumesneeded; /*volumes needed by job into system tables.*/
    newallocations(job)=newfiles; /*Newly catalogued entries due to this job.*/
    close(expandedjcl,'jclopenclosename');
    enqueue <job,expandedjclcatname> on scheduler for username;
        /*Now loop back to read the next job.*/
    go to discard;

                /*  ***************************  */
                /*          Job Statement Processing          */
                /*  ***************************  */

jobcard: if jobcardprocessed then go to endcard;;
    jobcardprocessed=true; /*signal job card means end of current job*/
        /*In the expanded JCL file, defaulted job parameters will be filled in by the following loop
        to their default values. In later examination of the job statement, we can assume that all
        parameters have been supplied.*/
    (∀y∈{'time','priority'} | image (y) eq Ω)
        image(y)=profile(username,y);
    end ∀y;
        /*Save job statement information in a local variable. If the JCL is free of errors, then the
        saved information will be used to update the system's tables.*/
    tempjobcard=image;
    if image('time') gt budget(username) then
            /*The estimated time exceeds the time allocation remaining for this user. We will send
            him an error message, continue to read his job for possible JCL errors, but not allow
            the job to execute.*/
        jclerror=true;
        expandedjob write 'Job rejected because the time estimate exceeds your
                current time allocation.';
    end if;
    go to mainloop;

                /*  ***************************  */
                /*          Job-Step Statement Processing          */
                /*  ***************************  */

exec:
    if image('proc') eq Ω then
        if image('label') is lbl ne nulc then
                /*test for duplicate step names*/
            if lbl+'.'+qualifierprefix∈stepnames then
```

```
            jclerror=true;
            expandedjcl write  '***duplicate step names - job rejected';
        else lbl+'.'+qualifierprefix in stepnames;
        end if lbl;
    go to mainloop;

else /*EXEC statement with proc parameter. Locate JCL macro procedure in a macro
       procedure library. We assume that every user, as well as the system, has a procedure
       library named 'proclib', which consists of JCL statements. First we make sure that a
       prog parameter is not also included.*/
    if image('prog') ne Ω then
        jclerror=true;
        expandedjcl write '***illegal "prog" field on a "proc" EXEC statement.';
else
    (∀y(i)∈<username,'system'>)  /*search user's library first*/
        if catalogue(<y,proclib>) ne Ω then
            /*jcl macro library exists -- generate a program file name, and search the
            library for the desired jcl macro.*/
            establish(<y,proclib>,macroopenclose,jclmacrofile);
            if procsearch(jclmacrofile,image('proc')) then
                /*The side effects of procsearch are that 'image' holds the first statement of
                the macro-expansion, and the jcl macro file is positioned to read the macro
                expansion.*/
                go to procfound;
            else unhook (macroopenclose);
            end if procsearch;
        end if catalogue;
    end ∀y(i);
    jclerror=true;
    expandedjcl write  '*** JCL macro procedure not found';
end if image;
go to mainloop;
procfound:
    qualifierprefix=lbl+'.'+qualifierprefix;
        /*In the next line -- currentposition is a macro*/
    currentposition=infile(2);  /*remember position in current file*/
    source=<<macroopenclose,jclmacrofile(2)>>+source;
    infile=jclmacrofile;  /*prepare to read from macro file.*/
    go to mainwrite;

                /*  ***************************  */
                /*          File Statement Processing          */
                /*  ***************************  */

filecard:
        /*A file statement before any exec statement is an error.*/
    if stepnames eq nl then
        jclerror=true;
        expandedjcl write 'File jcl statement before step statement.';
        go to mainloop; /*Ignore misplaced file card*/
    end if;
    progfilename=image('label');
        /*Check that a label, giving the program file name, is included.*/
    if progfilename eq Ω then
        jclerror=true;
        expandedjcl write '***Program file name not given in above line.'*/
        go to mainloop; /*quit processing this erroneous statement.*/
    end if;
```

```
userfilename=if image('name') eq Ω
        then newat /*scratch file-invent user file name*/
        else image('name');
    /*The fully qualified programfilename is the concatenation of the prefix and the program-
    filename given by the user.*/
fullprogname=qualifierprefix+'.'+progfilename;
    /*Check that the program file name is unique*/
if fullprogname ∈ jobfiles then
    jclerror=true;
    expandedjcl write '***Duplicate program file name in this step.'
    go to mainloop; /*quit processing erroneous statement.*/
end if;
if (∃c(i) ∈ userfilename | c eq '.') then
        /*External file name containing a '.' means that it is an indirect reference to a program
        file name in a previous step.*/
    if userfilename ∈ jobfiles then
        fullprogname in jobfiles;
            /*put fully qualified program file name, and associated user file name in expanded
            jcl file, to avoid recomputing the relationship between program and user file names
            during job control language interpretation. localfilemap maps fully qualified
            program file names into user file names. It is needed to handle indirect references
            to previously declared files (see section 3.2.3).*/
        localfilemap(fullprogname)=localfilemap(userfilename);
        expandedjcl write fullprogname,localfilemap(fullprogname);
    else jclerror=true;
        expandedjcl write '***reference to previously defined file
                    cannot be resolved.';
    end if;
    go to mainloop;
end if;


/*else, if not a reference to a previously declared file, */
fullprogname in jobfiles;
    /*If the data for the file follows this FILE statement, a blank atom will serve as it's user
    file name.*/
if userfilename eq '*' then userfilename=newat;;
    /*Note that the construction in the following statement, and the definition of the data file
    statement of section 3.2.3 do not provide for one user to access files of another user. The
    richer command language introduced in Chapter V remedies this deficiency.*/
localfilemap(fullprogname)=<username,userfilename>;
expandedjcl write fullprogname,localfilemap(fullprogname);


    /*If the file is already catalogued, parameters absent in the JCL statement are filled in
    from the catalogue. Later, one needn't worry about absent parameters on the JCL
    statement.*/
if catalogue(<username,userfilename>) ne Ω then
    (∀s ∈ {'device','disp','space'})
        if image(s) eq Ω then
            image(s)=catalogue(<username,userfilename>)(s);
        end if;
    end ∀s;
        /*Force volume specification to agree with catalogue.*/
    image('volume')=catalogue(<username,userfilename>)('volume');
else
        /*For a new file, parameters not specified in JCL are taken from defaults. If the user
        has not specified defaults for volume or space (the usual case), then the allocate routine
        will find space for the file.*/
    (∀s ∈ {'device','space','volume','disp'})
        if image(s) eq Ω then image(s)=default(username)(s);;
```

```
            catalogue(<username,userfilename>)(s)=image(s);
                /*In order to be able to drop newly catalogued files in the event that a job aborts or
                does not even pass input spool's JCL validity checking, we keep track of such file
                names.*/
            <username,userfilename> in newfiles;
            end ∀s;
    end if;
        /*'allocate' checks that a volume is assigned. If none is, then it will find space, and
        indicate the volume in the catalogue.*/
    allocate (<username,userfilename>);
        /*Keep track of volumes needed for this job.*/
    catalogue(<username,userfilename>)('volume') in volumesneeded;
        /*If the job contains a data file, copy the file onto the disc.*/
    if image('name') eq '*' then
            establish(<username,userfilename>,newfileopenclose,newfile);
            newfileopenclose in tempfiles;
            if n readingprimarysource then  /*resume reading from primary source*/
                currentposition=tl infile;
                infile=<open primarysource,primaryposition>;
            end if;
        rd: infile read line;
            if line('command') ne 'end-of-file' then
                newfile write line; go to rd;
            end if;
            close(newfile,newfileopenclose);
            unhook(newfileopenclose);
                /*If we had been reading from a macro library, then we temporarily close the
                primary source, and continue reading from the library.*/
            if not readingprimarysource then /*continue from library*/
                primaryposition=tl infile; /*mark position in primary source*/
                infile=<open currentsource,currentposition>;
            end if not readingprimarysource;
    end if image;
    go to mainloop;

definef procsearch(a,x);

        /*This routine searches a JCL macro library for a JCL procedure named x. The library
        consists of JCL statements. An EXEC statement having a 'prog' parameter x is the first
        statement in the expansion of the JCL macro x. If such an EXEC statement is found,
        procsearch returns true, has the side effect of leaving the library positioned to read the next
        sequential record in the expansion of x, and sets the variable 'image' to the first record in
        the macro's expansion.*/

    (while n endoffile(a))
        a read y;
        if<y('command'),y('prog')> eq <'EXEC',x>
        then image=y;
            return true;;
    end while;
    return false;
    end procsearch;

end spooling;
```

scope 3 jobsequencing;

```
/*  ***************************  */
/*       JOB CONTROL INTERPRETER       */
/*  ***************************  */
```

/*'jobcontrol' is passed the JCL and output files for a new job. For each step, it establishes mappings between program file names and user file names, determines whether steps are to be run or skipped, and loads the necessary programs. On encountering a step, a second process with the same mover as the current mover of the job control interpreter is set up. The second process has less privilege than the job control interpreter, so that it cannot interfere with the operating system. It is this process that ultimately executes the job step. When all the JCL for the step has been read and when the necessary code has been loaded, the interpreter makes the second process dispatchable, and it waits for that process to terminate before starting to interpret the next job step.*/

```
jobmonitor: <input,outfile>=processparameter;
        /*Assign program file names to the job's input and output files, positioning the output file
        beyond that portion already written.*/
    establish(input,inputname,source);
    establish(outfile,outputname,output);
    output(2)=#(output(1))+1; /*position at end of output file.*/
    mov=hd CPUcontrol;
    timeleft(mov)=esttime(mov);
        /*Get the process identifier for the user program*/
    userprocess=<mov,userprogs(mov)>; /*See Scheduler for creation of 'userprogs'*/
    user=owner(mov);
        /*Initialize error return code to show that no previous error had occured. The variable
        'fault' is used by abend to communicate with the job control interpreter, and info extracts
        the cause of the termination (see 2.4.3.8).*/
    info(fault)=0;
startstep: stepstarted=false;
        /*'stepstarted' becomes true when a step statement is encountered. A subsequent step
        statement acts as an 'end' for the current step.*/
    waittime=0; /*Initialize device mounting time estimate.*/
reader: source read image;
        /*We know that the JCL is syntactically correct, for otherwise input spool would have
        aborted the job. Thus, we don't check for valid JCL statement types. Of course, this
        assumes perfect transmission between disc and main memory, and perfect recording on
        disc. The non-JCL items written by input spool always follow FILE statements, and are
        read during file statement processing.*/
jump:
    go to {<'end',end>, /*branch table for commands*/
        <'job',reader>, /*job card can be ignored*/
        <'exec',exec>,
        <'file',file>} (image('command'));
exec:
    if stepstarted then
            /*If we encounter two exec statements without an intervening end statement, the
            second exec ends the previous step. We must reposition the input file so that it will be
            re-read when JCL interpretation resumes. Backspace one record.*/
        source(2)=if (source(2)-2>∃n>1 | (hd source)(n) eq er) then n+1 else 1;
        go to end;
    end if stepstarted;
    if image('proc') ne Ω then
            /*Since JCL macros have been expanded by inputspool, ignore statements with 'proc'
            parameter.*/
        go to reader;
    end if image;
```

```
        /*save EXEC statement information in local storage for later use.*/
    progstobeloaded=image('prog'); /*Save name of program to be loaded.*/
    args=image('parm'); /*JCL input data to the job step*/
    steptimeleft(userprocess)=timeleft(mov) min image('time');
    output write image;
        /*Test condition on JCL statement to determine whether to run this step. The return code
        from the previous step is in info(fault). eval (not shown) checks the cond parameter
        against the error return code info(fault).*/
    if n eval image ('cond') then
                output write 'step skipped - condition false';
            skip: source read image;
                if image('command') ∈ <'file','end'> or image('command') eq Ω then
                    go to skip;
                else
                    go to jump; /*Next step found.*/
                end if image;
    else
        stepstarted=true;
    end if;
    go to reader;

file:
        /*Inputspool has inserted a second JCL file statement for every file, giving the fully
        qualified program file name and user file name.*/
    source read pfn,ufn;
        /*The disposition of a file may change from step to step (e.g. the last step to use a tempo-
        rary file should specify NOCAT to drop the file from the system's catalogue.*/
    catalogue(ufn)('disp')=image('disp');
    ufn out newallocations(mov); /*New file no longer unused.*/
    reserve(allocflag); /*Prevent other allocation action.*/
    if not(∃dselected ∈ devices | mounted(dselected) eq image('volume')) then
            /*If volume is not mounted, select an appropriate device and estimate mounting time at
            1 min. Normally, the scheduler has determined that enough devices are available, so
            that the following statement will succeed. If one wishes to take device failure into
            account, more care will be needed below.*/
        dselected=arb{d,d ∈ devices | type(d) eq image('device') and mounted(d) eq nl};
        mounted(dselected)=image('volume');
        dselected out ready; /*device is not ready until operator says it is.*/
            /*Send console message to the operator to mount the appropriate volume on the
            chosen device. It is understood that he will dismount whatever is currently mounted on
            device dselected.*/
        operatormessage
            ('mount'+image('volume')+'on device',dselected,'for user'+user);
        waittime=waittime+oneminute;
            /*Even if this step requires a volume mounting, we continue to interpret JCL until we
            reach the end-of-step, in case additional volume mounting requests are discovered for
            this step.*/
    end if;
    catalogue(ufn)('deviceaddress')=dselected;
        /*Associate program file name with user file name*/
    assign(<userprocess,image('label')>,ufn,standardfixup);
    free allocflag; /*Allocation completed.*/
    go to reader;

        /*An 'end' statement or an 'exec' statement bring control here. If there is no step started,
        then the end card represents the end of job. Otherwise, it is time to check that all I/O is
        ready, and to load the program.*/

end:  if n stepstarted then go to endjob;;
```

```
    /*If there is a setup delay, try to find a non-setup job to run during the delay.*/
  if waittime gt 0 then
    quickrun=true; /*Signal scheduler to try for non-setup jobs.
      Enqueue the expected time for setup completion on the scheduler. The scheduler will
      use this time as an upper bound for the completion time of short runs which it will try
      to run to overlap the set up time for this job.*/
    enqueue(waittime+clock) on scheduler for scheduler;
    (∀x∈devwait) await x∈ready;;
      /*Now all devices are ready. Signal scheduler to cease looking for non-setup runs.*/
    quickrun=false;
      /*Wait for non-setup job to finish, if one is running. 'userprogs' is the set of user-
      processes which the scheduler has set up and which have not concluded. Normally, this
      is a singleton set consisting of the user process controlled by the job control analyzer
      process. But while setup is being overlapped with a non-setup job, there are two items
      in 'userprogs'. Waiting for 'userprogs' to become 1 is the means by which this job
      control process waits for a non-setup job to complete. The scheduler will not attempt
      to start any additional short jobs, since 'quickrun' has been set to false.*/
    await #userprogs eq 1;
  end if;
    /*We go on to load the routines needed for the step which is now to be carried out.*/

/*          Loader          */

    /*'progstobeloaded' holds names of programs to be loaded for the current step. 'fetch' is a
    machine dependent routine that resolves interprogram references and which places in
    'missing' the names of referenced programs which have not yet been loaded.*/
loadloop: missing=progstobeloaded;
    /*Initialize 'loadertable', the table used by the loader to store subprogram entry point
    locations, and 'codevect', the bit string which the loader will build into the executable
    program.*/
  loadertable=nl;
  codevect=nult;
  (while missing ne nl doing progstobeloaded=missing;)
    (∀x∈progstobeloaded)
      (∀y(i)∈<user,'sys'>)
        if catalogue(<y,proglib+'.'+x> is c) ne Ω then
          x out missing;
            /*'fetch' (not shown here) moves machine language text for the program x
            into the appropriate portion of memory, and extablishes communication
            between x and previously loaded programs. If x requires other programs to
            be present, 'fetch' puts these program names into 'missing'. 'fetch' stores
            the entry point into 'main', unless there is not sufficient room for the pro-
            gram, in which case 'main' is set to Ω. A table of entry points for all subpro-
            grams is built in 'loadtable'. In the uniprogrammed case, we assume that the
            loader has been initialized with the size of the largest allowable program
            string. 'codevect' is built into the executable program by the loader.*/
          fetch (c,missing,main,codevect,loadertable);
          continue ∀x;
        end if;
      end ∀y(i);
    end ∀x;
    if missing*progstobeloaded is nf ne nl then
        /*Some required programs can't be found!*/
      output write 'missing routines:'+nf+'. Step skipped.';
      info(fault)=notloaded; /*notloaded is given some numeric value.*/
      go to stepskipped; /*Treat as an abnormal termination.*/
    end if;
  end while;
```

```
        /*Test whether there was enough room for the program to be loaded.*/
    if main eq Ω then
        output write 'Not enough space to load this job step.'*/
        info(fault)=notloaded;
        go to stepskipped;
    end if;

        /*If and when the program is fully loaded, The new user process is given a standard
        environment, to which 'standardenvironment' is assumed to be initialized, and whose
        description is machine dependent. We assume that 'fetch' has placed the executable
        program in the appropriate area in storage, starting at 'origin', and that the entry point was
        computed by 'fetch' and stored in 'main'. initialstate is set to the initial state for the user
        process.*/
    (disable)
        loctr(standardenvironment)=main; /*main entry computed by fetch*/
        code(standardenvironment)=codevect; /*Moves loaded code into environment*/
        initialstate=<userprocess,standardenvironment>;
            /*Assume that the entire step time estimate will be used.*/
        timeleft(mov)=timeleft(mov)-steptimeleft(userprocess);
        budget(user)=budget(user)-steptimeleft(userprocess);
        split to initialstate(args) for thisprocess; /*start execution of job step.*/

/*          End of Step          */

        /*Await execution termination report from 'abend'. In response to an abend monitor-call,
        the numeric code for the termination gets queued on the job control interpreter process.*/
    await getfirst(thisprocess) is fault ne Ω;
        /*A normal termination or abend will satisfy the above wait condition. Restore the
        balance of the step's time to the time remaining for this job.*/
    timeleft(mov)=timeleft(mov)+steptimeleft(userprocess);
    budget(user)=budget(user)+steptimeleft(userprocess);
stepskipped:
    if info(fault) ne 0 then
            /*Abnormal end -- write termination message.*/
        output write '*** step stopped', info(fault);
    end if timeleft;
    kill userprocess; /*In any case, remove the user process*/
    (∀files∈programfiles(userprocess))
            /*release all programfiles for the completed job step.*/
        unhook (files); /*If the file's disposition was not
                'keep', then space for the file will be released by unhook.*/
    end ∀files;
    if timeleft(mov) le 0 then
            /*The job has used up all of it's estimated running time.*/
        output write 'Time estimate exceeded.';
        go to endjob;
    end if;
        /*Even if a step abends, the run may continue, as determined by the 'cond' parameter of
        the next 'exec' statement.*/
    go to startstep;

        /*At the end of a job, the output file is passed to output spool.*/
endjob: close (hd output,outputname);;
    unhook(outputname);
    enqueue outfile on outspool for mov;
    unhook(inputname);
    (∀y∈newallocations(mov))
            /*Remove from the catalogue, all new files which have not been used during the
            execution of this job.*/
```

```
        relinquish(y); /*give up space.*/
        catalogue(y)=Ω;
    end ∀y;
    userprocess out userprogs;
    resources(mov)=Ω;
        /*Items enqueued on the scheduler from job control interpreters signify that a user job has
        terminated. The scheduler will respond by attempting to schedule another job.*/
    enqueue nl on scheduler for thisprocess;
    term;
        /*This job control analyzer process is now completed and thus terminates itself.*/


                /*  ***************************  */
                /*            SCHEDULER          */
                /*  ***************************  */

    /*The scheduler is enqueued when:
        a) input spool has read a new job, when
        b) jobcontrol terminates a user job, and when
        c) jobcontrol seeks work to overlap setup time.*/
schedule: await #workset{thisprocess})>0;
    request=getfirst(thisprocess);
        /*The queue on the scheduler is the communication medium between the scheduler and its
        callers. When the ancestor of an enqueued item is a user, a new job has just been read, and
        is placed at the end of the vector 'backlog', which contains all jobs to be run in the order
        received (except for rearrangements due to operator-issued priority commands.*/
    if ancestor(request) is requestor∈ users then /*true for case a*/
        if (∃x(i)∈backlog | priority(x)<priority(hd info(request))) then
                /*insert new item after all jobs having the same or higher priority but ahead of jobs
                having lower priority*/
            backlog(i:)=<info(request)>+backlog(i:);
        else backlog(#backlog+1)=info(request);
        end if;
    end if ancestor;

    if requestor eq thisprocess then
            /*True, by convention, for case c, c.f. the lines of code following the label 'end' in the
            job control interpreter, above. The scheduler uses the identification of its caller to
            determine which of the above cases hold. Case c arises when the scheduler itself is the
            requestor, case a arises when a user is the requestor, and all other cases are case b.*/
        timelimit=info(request); /*record time before which job must end*/
    end if;
    if quickrun and #userprogs eq 1 then
            /*Since (#userprogs) eq 1, no non-setup job has yet been scheduled, and therefore we
            look for a short non-setup job.*/
        if(∃x(i)∈backlog, ∀r∈resources(x(i)), ∃d∈devices*ready |
                (mounted(d) eq r) and (esttime(x(i,1))+clock le timelimit) then
                    /*a job exists with estimated running time less than the estimated setup time,
                    and with all resources ready, i.e. no mounting time.*/
            <jobid,input>=x;
            backlog(i:)=backlog(i+1:);
            go to startup;
        else
            go to schedule;
        end if (∃x;
    end if quickrun;
    /*cases a and b reach this point*/
nextjob:
```

```
    if (#userprogs gt 0) or #backlog eq 0 then go to schedule;;
            /*If a user program was running, then uniprogramming discipline prevented further
            scheduling at this time.*/
    <<jobid,input>,backlog>=<backlog(1),backlog(2:)>;
        /*First job on scheduler's queue is now selected.*/
    assign(newat is file,input,nl);
    if notenough(jobid) then
        monitor(rewind,file);
        monitor(write,file,'*** insufficient number of devices to run job');
        enqueue input on outspool for schedule;
        go to nextjob;
    end if;

/*          Job Selected          */

        /*Define an output file for the job which has been scheduled to run.*/
startup: catalogue(<jobid,newat> is output)=
                        {<'device',tempdisc>,<'space',filesize>,
                         <'disp',{'cat','leave'}>};
    allocate (output);
        /*Set up process identifier for the user's programs.*/
    userprogs(jobid)=newat;
        /*Having just catalogued the job's output file, a job control process is started which loads
        job steps and links program file names to user file names.*/
    split to <<jobid,newat>,jobmonitor>(<input,output>) for thisprocess;
    go to schedule;

definef notenough(job);
        /*Returns true if there are not enough tape or disc drives to run this job.*/

    (∀t∈{'tape','tempdisc'})
        ndev=#{v∈resources(job) | hd (volumes(v)) eq t};
        if ndev gt #{d∈devices | type(d) eq t} then
            return true;
        end if;
    end ∀t;
    return false;
end;

end jobsequencing;
```

```
/*  ***************************  */
/*          OUTPUT SPOOL         */
/*  ***************************  */

scope 4 ospooling;

        /*Output spool illustrates the use of the facilities described in section 3.4.1.7 for perform-
        ing I/O. The SETL I/O could have been used instead, and the main loop would then be:
                pfn read line;
                (while line ne Ω)
                    print line;
                    pfn read line;
                end while;
```

Instead, we indicate the way that a fairly conventional double buffering scheme can be represented in our language.

Each process which prints system output files is first passed the name of a file which already corresponds to a physical printer.*/

```
outputspool: device=processparameter;
   <buf1,buf2,intbuf>=<newat,newat,newat>; /*Blank atoms to point to I/O buffers.*/
   assign(<thisprocess,newat> is printer,device,printerfixup);
top: await (getfirst(thisprocess) is ufn) ne Ω; /*wait for work*/
   assign(<thisprocess,newat> is pfn,ufn,discfixup); /*Relate a program file name to the file to
      be printed, and indicate that this process has an I/O fixup routine at 'discfixup' to which
      control will be forced at the end of each disc operation.*/
   useleftbuffer=true; /*initialize buffer switch*/
   eof=false; /*initialize end of file switch*/
   monitor('read',pfn,buf1); /*read initial record*/
      /*The above monitor request started a read operation. When it is physically complete,
      control is forced to discfixup. In this way, action can be taken on the physical completion
      of an I/O operation. In this program, the action is generally to start to read the next
      record, while printing the record just read. A standard double buffering scheme is used.*/
   nconsecreaderrors=0; /*initialize reading error count*/
      /* The following idle loop relinquishes control of the CPU (unless an end of file has
      occured). Control is forced to discfixup when the end of a disc operation occurs.*/
idle: monitor('wait',pfn);
      /*discfixup will set eof to true when the end of file has been reached. The printer is spaced
      to a new page, and another request for printing is awaited.*/
   if n eof then go to idle;;
   monitor('wait',printer);
   monitor('space',printer); /*space printer to new page*/
   unhook(pfn); /*release user's output file*/
   go to top; /*look for more work*/

            /* ************************** */
            /*          Fixup Routines    */
            /* ************************** */

      /*All the real work is done in the fixup routines. Control normally does not reach main-
      stream until the end of the disc file has been encountered. The waits in the fixup routines
      prevent returns to mainstream, for on satisfying the wait, control is forced to another fixup
      routine.*/

            /* ************************** */
            /*          Disc Fixup        */
            /* ************************** */

discfixup:
      /* 'endoffile' is a macro which is true if end of file is indicated in the I/O interrupt bits. It
      is a machine dependent macro.*/
   monitor('iointerrupt',intbuf); /*Determine cause of I/O termination.*/
   int=buffer intbuf;
   if endoffile(int) then
      eof=true; /*file finished*/
      monitor('endfixup'); /*return to mainstream*/
   end if;

      /*'ok' is a machine dependent macro which is true if the I/O interrupt bits do not indicate
      an error.*/
```

```
    if n ok(int) then  /*disc error fixup follows*/
        (while nconsecreaderrors lt 10)
            nconsecreaderrors=nconsecreaderrors+1;
            monitor('backspace',pfn);
            monitor('release',pfn);  /*presume backspace ok, ignore interrupts*/
            monitor('wait',pfn);
                /*'useleftbuffer' determines into which buffer to read.*/
            if useleftbuffer then
                monitor('read', pfn, buf1);
            else
                monitor('read', pfn, buf2);
            end if;
            monitor('wait', pfn); /*wait for reread, then re-execute discfixup*/
        end while;
        if useleftbuffer then  /*print error message on unreadable record*/
            buffer buf1='**unreadable record**';
        else buffer buf2='**unreadable record**';
        end if;
    end if n ok;
    useleftbuffer= n useleftbuffer; /*switch buffers*/
    nconsecreaderrors=0; /*Previous errors are now forgotten.*/
    monitor('wait',printer);
        /* Note:  If the printer is not busy, which is the case after the initial read, control falls thru
        to the printer fixup anyway, and the most recently read disc record is printed.*/

                /*  ***************************  */
                /*            Printer Fixup      */
                /*  ***************************  */

printerfixup:
    if eof or count gt 0 then monitor(endfixup);;
            /*Read into the buffer whose contents had just been printed out.  Then print the
            contents of the other buffer into which a disc read had just been completed.*/
    if readleftbuffer then
        monitor('read',pfn,buf2);
        monitor('write',printer,buf1);
    else
        monitor('read',pfn,buf1);
        monitor('write',printer,buf2);
    end if;
    monitor('wait',pfn); /*Prevents printer interrupt being processed before disc.*/
    go to discfixup;

end ospooling;



                /*  ***************************  */
                /*    OPERATOR MESSAGE ANALYZER  */
                /*  ***************************  */

scope 2 allocation;
        /*This process acts on operator generated messages.  The process waits to read a message,
        and then executes a short block of code to satisfy the request.
```

```
        It is also the first section of code executed when the operating system starts running, so
        that it initializes the interrupt handlers and the scheduler.*/

initially
    (disable)
            /*Generate a process state with full privileges by copying the state of the current
            process. This will be used to generate the initial states of the interrupt processes.*/
        osenvironment=state(thisprocess);
            /*Set up timer-interrupt handler*/
        loctr(osenvironment)=timexpt;
        interrupt(timer)=osenvironment;
            /*Set up monitor-call interrupt handler.*/
        loctr(osenvironment)=monitorxpt;
        interrupt(monitor)=osenvironemnt;
            /*Set up I/O interrupt handler.*/
        loctr(osenvironment)=ioxpt;
        interrupt(io)=osenvironment;
            /*Start the scheduler.*/
        loctr(osenvironment)=schedule;
        schedstart=<<moverpart(thisprocess),newat> is scheduler, osenvironment>;
        split to schedstart for thisprocess;
            /*Start the dispatcher process*/
        loctr(osenvironment)=getwork;
        dispstart=<<moverpart(thisprocess),newat> is dispatcher, osenvironment>;
        split to dispstart for thisprocess;
    end disable;
        /*Open the operator's console keyboard for I/O.*/
    establish<<sys,'keyboard'>,keyboardopen,keyboard);
end initially;

        /*Now, read from the console keyboard. An eventual start command will get input spool
        working.*/

operation: keyboard read msg;
    if msg('command') ∈ hd ({<'start',start>,
                <'add',addnewuser>,
                <'mounted',readymounted>,
                <'givetime',addtime>,
                <'drop',dropuser>,
                <'priority',hipriority>} is list) then
        go to list(msg('command'));
    else
            /*illegal command*/
        keyboard write 'illegal command';
        go to operation;
    end if;

        /*The operator's commands,
            {<'command','start'>,<'device',d>,<'action','cardin'>}
            or {<'command','start'>,<'device',d>,<'action','printer'>}
        designate device d as a system input or a system output device. These commands are used
        to initiate spooling.*/
start:
        /*The 'type' table classifies devices by type (See 3.5.1.1). The device type of uf will be
        either 'cardin', or 'printer'. The type of uf must agree with the 'action' parameter.*/
    if type(msg('device')) eq msg('action') /*and device is idle*/
            and (∀p ∈ programfiles | deviceaddress(p) ne msg('device')) then
        <'sys',newat> is uf in datafiles;
```

```
        catalogue(uf)={<'deviceaddress',msg('address')>,
                       <'device',type(msg('device'))>};
        split to (if msg('action') eq 'cardin' then inputspool
                  else outspool) (uf) for CPUcontrol;
    else keyboard write 'inappropriate device';;
    go to operation;

        /*The command {<'command','add'>,<'user','u'>,<'profile',<...>>} adds u to the set of
        system users, and establishes u's defaults as given in the profile parameter.*/
addnewuser:
        /*Here, we check that msg('profile') is a tuple of pairs of character strings, and reject the
        command if the profile is of the wrong form. In actual practice, a more precise check
        would be desireable.*/
    if type msg('profile') eq tupl and
        (∀t(n)∈msg('profile'), e∈t |
            type t eq tupl and #t eq 2 and type e eq cstring) then
        msg('user') is u in users;
        default(u)=msg('profile');
    else
        keyboard write 'Bad format for defaults. Command rejected.';
    end if;
    go to operation;

        /*The command {<'command','priority'>,<'user','u'>} elevates the first of user u's jobs
        in the backlog to be the first job in the backlog.*/
hipriority:
    (disable) /*to avoid race condition with the scheduler*/
        if (∃x(i)∈backlog | owner(hd x) eq msg('user')) then
            backlog=x+backlog(1:i-1)+backlog(i+1:);
            priority(hd x)=9;  /*Highest priority*/
        end if;
    end disable;
    go to operation;

readymounted:
        /*The command, {<'command','mounted'>,<'volume',v>,<'device',d>} communicates
        to the system that the volume v has been mounted on device d by the operator, and that it
        is ready for use. The system must now complete any space allocation on this volume that
        was left incomplete by the 'allocate' routine. Since allocation may be done as a result of
        this command, 'allocflag' must first be reserved. (See the storage allocation routines.)*/

    reserve(allocflag);
    if mounted(msg('device') is d) eq msg('volume') then
            /*The operator has mounted the volume which the operating system has been expect-
            ing. Now complete any incomplete allocations, and then indicate that the device is
            ready for use.*/
        if type(d) eq 'tempdisc' then /*Complete allocation of disc files on this volume*/
                /*Note that 'deviceaddress' is a macro defined at the beginning of Section 3.5.3.*/
            (∀pfn∈programfiles | deviceaddress(pfn) eq d
                        and catalogue(userfile(pfn))('extents') eq Ω)
                    /*Lack of extent field in catalogue entry for the file indicates that allocation of
                    the file is incomplete.*/
                if spacefound(d,userfile(pfn)) then
                    continue ∀pfn;
                /*else there is an error in the use of the disc pack, because the volume table of
                    contents is inconsistent with the system table, 'volumes'. Note that
                    'spacefound' (code for which is shown below) updates the catalogue to show
                    the newly assigned file 'extents'. 'allocate' had earlier decided that sufficient
```

```
                space existed on this volume. The resolution of this problem is not shown
                here.*/
            end if;
        end ∀pfn;
    end if type(d);
    msg('device') in ready;
else if mounted(msg('device')) ne Ω then
        /*The operator has mounted the wrong volume. Inform him of the correct volume to
        be mounted.*/
    operatormessage('Mount volume'+msg('volume')+'on device'+msg('device'));
else
        /*The operator has mounted a volume on a device which did not hold a required
        volume. Mark the device as being ready.*/
    msg('device') in ready;
end if mounted;
free allocflag;
go to operation;

addtime:
    /*The command, {<'command','givetime'>,<'user',u>,<'time',t>} allocates an additional
    t units of time to user u.*/
(disable)
    if msg('user') is u ∈ users then
        budget(u)=budget(u)+msg('time');
    else
        keyboard write 'improper user specified.';
    end if;
end disable;
go to operation;

dropuser:
    /*The command, {<'command','drop'>,<'user',u>} causes the user u to be removed from
    the set of authorized system users. Any work in progress for u is purged, as well as system
    structures related to u.*/
if msg('user') is u ∈ users then
    (∀p∈processes | owner(hd p) eq u))
            /* Remove all active processes associated with user u. First, make sure that all
            user files are detached, and not involved in I/O operations (done by subroutine
            'unhook'.)*/
        (∀files∈programfiles(p))
            unhook(files); /*Stop I/O action, disconnect file from process.*/
        end ∀ files;
        kill p;
    end ∀p;
    catalogue{u}=Ω;
    defaults(u)=Ω;
    datafiles(u)=Ω;
    maxprio(u)=Ω;
    budget(u)=Ω;
    u out users;
else
    keyboard write 'Improper user specified.';
end if;
go to operation;

end operator;
```

```
/*  ***************************  */
/*          FILEMAP MAINTENANCE    */
/*  ***************************  */

define assign(programfilename,userfilename,fix);
        /*This subroutine establishes 'programfilename' as the program file name for the user
        file 'userfilename', and specifies 'fix' as the location of programfilename's interrupt
        fixup.*/
    programfilename in programfiles;
    userfile(programfilename)=userfilename;
    fixup(programfilename)=fix;
    return;
        /*Initialize the maps used for monitor services relating to I/O on this file. (cf. sections
        3.5.1.5 and 3.5.2.7)*/
    disabled(programfilename)=false; /*File to cause logical interrupts.*/
    rel(programfilename)=false; /*Interrupt information is not to be discarded.*/
    filewait(programfilename)=false; /*Program is not waiting for I/O on file to end.*/
end;

        /*This subroutine undoes the effects of 'assign' and uncatalogues the file if necessary.*/
define unhook(programfilename);
        /*If the program file, programfilename, has an uncompleted operation in progress, prevent
        an interrupt from going to the user-program's fixup, by establishing the system's standard
        fixup as the file's interrupt handler.*/
    monitor('fixup',programfilename,standardfixup);
    monitor('wait',programfilename);
    if n ('cat'∈(catalogue(userfile(programfilename))('disp'))) then
        relinquish(userfile(programfilename)); /*Give up space on volume.*/
        catalogue(userfile(programfilename))=Ω; /Delete file from catalogue.*/
    end if;
        /*Indicate that the unhooked file is no longer active on the physical device on which it
        resides. If no program files are assigned to the device, indicate that the device is not
        logically mounted.*/
    if ∀pfn∈programfiles | catalogue(userfile(pfn))('volume') ne mounted(d) then
        mounted(d)=Ω;
    end if;
        /*Note that 'deviceaddress' is a macro. See the start of Section 3.5.3.*/
    deviceaddress(programfilename)=Ω;
    userfile(programfilename)=Ω;
    fixup(programfilename)=Ω;
    disabled(programfilename)=Ω;
    rel(programfilename)=Ω;
    filewait(programfilename)=Ω;
    file out programfiles;
    return;
end;

/*  ***************************  */
/*    EXTERNAL STORAGE ALLOCATION  */
/*  ***************************  */

define qd allocate(userfilename) on allocflag;
        /*This routine assigns a disc pack or tape reel to a file. This routine, as well as the
        following one, is queued to prevent simultaneous attempts to allocate or free space on
        devices by different portions of the operating system. We assume that allocflag is an atom
```

```
        which has been made a member of facilities. If a volume is already assigned, there is
        nothing to do.*/

if catalogue(userfilename)('volume') ne Ω then go to quit;;
if catalogue(userfilename)('device') ∈ discs then

            /*Disc space allocation*/
    if catalogue(userfilename)('space') eq Ω then
            /*assign a size to the file if not already done. Note that 'filesize' is a macro-name.*/
        catalogue(userfilename)('space')=filesize;
    end if;

        /*Search mounted packs for available space.*/
    (∀d ∈ devices | type(d) eq catalogue(userfilename)('device'))
            /*Try to find sufficient space for the file on the volume mounted on device d.*/
        if spacefound(d,userfilename) then
                /*Note that 'spacefound' updates operating system's table of space free on vol-
                ume.*/
            go to quit;
        end if;
    end ∀d;

        /*Having failed to find space on any volume mounted on a disc drive, we now search for
        any dismounted pack having sufficient space. If such a volume is found, assignment of
        space will be made later when the volume is mounted. The request for mounting will come
        from the JCL interpreter.*/
    if (∃v ∈ volumes | v(2) eq catalogue(userfilename)('device') and
                        v(3) ge catalogue(userfilename)('space')) then
        catalogue(userfilename)('volume')=v(1);
        volumes(v(1),v(2))=v(3)-catalogue(userfilename)('space');
    end if;
    go to quit;

        /*Tape volume assignment.*/
else if catalogue(userfilename)('device') eq 'tape' then
            /*First, search the tape drives to see if an unassigned reel is
            mounted. If one is, it is assigned to the file.*/
    if (∃d ∈ devices | type(d) eq 'tape' and
                        volumes(mounted(d),'tape') eq 'empty') then
        catalogue(userfilename)('volume')=mounted(d);
        volumes(mounted(d),'tape')=userfilename;
        catalogue(userfilename)('deviceaddress')=d; go to quit;
    end if;
        /*If no mounted, unassigned tape reel is found, a dismounted
        reel is assigned. Request for mounting will be issued by the
        JCL interpreter.*/
    if (∃v ∈ volumes | v(2) eq 'tape' and v(3) = 'empty') then
            catalogue(userfilename)('volume')=v(1);
            volumes(v(1),v(2))=userfilename;
    end if;
end if;

quit:
    free allocflag; /*Make allocate and relinquish routines available.*/
    return;

end allocate;
```

```
define qd relinquish(userfilename) on allocflag;
        /*This routine alters the space map on a disc to show that the space for a discontinued file
        is available for assignment*/

    d=(catalogue(userfilename) is item)('deviceaddress');
    establish(d,tocopenclose,toc);
    toc read volname,map;
    map(userfilename)=Ω; /*space no longer allocated to file*/
    <'empty',item('extents',1),item('extents',2)> is t in map;
        /*Previously allocated space is now empty. If two or three empty regions are now contigu-
        ous, combine them into one large empty region.*/
    if ∃g∈map | (g(1:2) eq <'empty',t(2)+t(3)>) then
        g out map;
        t out map;
        <'empty',t(2),t(3)+g(3)> is t in map;
    end if;
    if ∃g∈map | (g(1) eq 'empty' and t(2) eq g(2)+g(3)) then
        g out map;
        t out map;
        <'empty',g(2),t(3)+g(3)> in map;
    end if;
    toc(2)=1; /*Rewind the table of contents file*/
    toc write volname,map; /*and write out the updated table of contents.*/
    close(toc,tocopenclose);
    unhook (tocopenclose);
        /*Indicate in the system's tables the largest free space that exists on this volume.*/
    volumes(volname,type(d))=[max: g∈map | g(1) eq 'empty'] g(3);
    free allocflag;
    return;
end relinquish;

definef spacefound(d,userfilename);
        /*This function returns true if the volume mounted on device d has sufficient space to hold
        file userfilename. If there is sufficient space, then the volume table of contents are updated
        to show the area on the pack which has been assigned to the file. If there is not sufficient
        space, then the routine returns false.*/

    if n (d∈ready) then return false;;
    establish(d,tocopenclose,toc);
    toc read volname,map;
        /*We assume that all volumes start with a table of contents, which consists of a character
        string read into 'volname', giving the volume's identification, followed by a set read into
        'map', which consists of triples, <file,start,length>, where 'file' is a user file name, 'start'
        its starting position on this volume, and 'length' is the size of the area. A triple with first
        component 'empty' describes available space.*/
    if not (∃x∈map | x(1) eq 'empty' and
                    x(3) ge catalogue(userfilename)('space')) then
        return false;
    else
        x out map;
            /*If the space found is too large, indicate that the remaining space is still available.*/
        if x(3)-catalogue(userfilename)('space') is t gt 0 then
            <x(1),x(2)+x(3)-t,t> in map;;
        <userfilename,x(2),catalogue(ds)('space')> in map;
        toc(2)=1; /*Rewind space-map*/
        toc write map; /replace space-map*/
        close(toc,tocopenclose); /*with updated version*/
        unhook(tocopenclose);
        catalogue(userfilename)('volume')=volname;
```

```
        catalogue(userfilename)('deviceaddress')=d;
        mounted(d)=volname;
        <x(2),catalogue(userfilename)('space')> in catalogue(ds)('extents');
        volumes(volname,type(d))=[max: x∈map | x(1) eq 'empty'] x(3);
        return true;
    end if;
end spacefound;

end allocation;



        /*  ***************************  */
        /*          MONITOR SERVICES     */
        /*  ***************************  */

    /*Control goes to monitorxpt on an interrupt caused by a request for monitor service. A
    new disabled process handles the interrupt by spawning an enabled process to handle the
    request, after which the disabled process terminates itself. See also, section 3.3.*/

macro filebeginning; /*extracts position on disc where file starts*/
catalogue(file)('extents',1)
endm filebeginning;

macro waitnotbusy(d); /*By re-reserving busystatus(d), the monitor*/
    reserve(busystatus(d)) /*request handler effects a 'wait' until d has */
endm waitnotbusy; /*finished it's I/O operation.*/

macro maxposition; /*extracts maximum file size*/
catalogue(file)('extents',2)
endm maxposition;

macro filepart(parm); /*Extract program file identifier from arguments to monitor services.*/
    parm(1)
endm filepart;

macro buffpart(parm); /*Extract buffer to be used for monitor service operation.*/
    parm(2)
endm buffpart;

macro labelpart(parm); /*Extract location of fixup routine from arguments.*/
    parm(2);
endm labelpart;

monitorxpt:
    (disable)

        /*For an overview of the code which follows, cf. section 3.5.2.7.*/
        /*First, keep track of time used by problem program.*/
    if resume∈ userprogs then
            /*Compute the time remaining in the user's jobstep, and charge a fixed rate,
            'overhead', for the monitor service. Alternatively, we could compute the actual
            time to perform the monitor service, and charge for it on completion. 'overhead'
            should be defined by a macro to be the appropriate number of clock ticks to charge
            for the average monitor service (often on the order of 1 ms).*/
```

```
            steptimeleft(resume)=timer - clock - overhead;
        end if;

            /*Having received a monitor call, we split off a new process which will follow the
            requested monitor service through to its end, and then at once set up to receive the next
            monitor call. We assume that 'cause' is of the form <fcn,parm>, where fcn specifies
            the monitor function and parm gives any additional parameters. Recall that
            'osenvironment' represents an operating system environment, including system privi-
            lege.*/
        loctr(osenvironment)=lookatrequest; /*'lookatrequest' is a label
                        occurring a few lines below.*/
            /*Start a privileged process to interpret the monitor service request. Attach the
            process to the mover of the requesting process.*/
        state(<moverpart(resume),newat>is newprocess) = osenvironment;
        split to newprocess(cause) for resume;
            /* We cheat the dispatcher in what follows, by dispatching the newly spawned process
            ourselves to replace the disabled monitor interrupt handler, and at the same time, we
            force the monitor interrupt process to start at the head of the disabled block the next
            time it becomes activated.*/
        remove(CPU,newprocess); /*Now the dispatcher can't act on 'newprocess'*/
        <CPUcontrol,loctr(state(thisprocess))>=<newprocess,monitorxpt>;
end disable;

        /*The newly spawned process which handles the monitor request starts here.*/
lookatrequest:
    <fcn,parm>=info(thisprocess); /*save parameters*/
    caller=ancestor(thisprocess); /*save id. of calling process*/
    if  {<'read',io>,  /*All I/O requests go to*/
        <'write',io>,  /*the label 'io' first.*/
        <'backspace',io>,
        <'rewind',io>,
        <'space',io>,
        <'wait',io>,
        <'enable',io>,
        <'disable',io>,
        <'release',io>,
        <'endfixup',endfixup>,
        <'iointerrupt',giveinterruptcause>,
        <'endstep',endstepaddr>
        <'fixup',fixupaddr>,
        <'abend',setabend>}(fcn) is loc ne Ω then go to loc;;

        /*Illegal monitor call. Let control flow through to the abend monitor call routine, with
        'parm' indicating termination due to an illegal monitor call. (Assume that the variable
        'badparameter' indicates this reason for termination.*/
    parm=badparameter;
    go to setabend;

endstepaddr: parm=0;  /*Code to indicate normal completion.*/
setabend: /*Determine the ancestor of the abending process, and put the cause of termination on
        the ancestor's work queue. (Recall that the ancestor is the job control analyzer process
        which had set up the user job, and awaits a termination report.) This action will awaken the
        ancestor which will terminate any I/O which might still be in progress for the terminating
        process, and which will start the next job step or terminate the job.*/
    jcinterpprocess=ancestor(initialvar(state(caller)));
    enqueue parm on jcinterpprocess for nl;
    term;

io:
```

```
file = <caller,filepart(parm)>; /*The name in program file name space is unique because it is
    the pair formed by the unique process identifier and the program-generated name. Thus,
    several processes using the same program name at the same time, will be able to access
    separate files. See the construction of the 'userfile' structure during file statement process-
    ing in the job control interpreter.*/
if not(file ∈ programfiles) then
        /*I/O request for a non-existent program file. Abend the calling process. Assume that
        the variable 'badfileid' indicates this reason for termination.*/
    parm=badfileid;
    go to setabend;
end if not;
iodevice = deviceaddress(file); /*device on which file is mounted. See 3.5.1.1 and 3.5.1.5,
    and the macro definition for deviceaddress.*/
chan = arb{c ∈ channels | iodevice ∈ units(c)}; /*Channel used for the file.*/
    /*Control functions that do not require the ability to physically manipulate an I/O device
    can now be executed.*/
if  {<'wait',wait>,
    <'release',release>,
    <'enable',enable>,
    <'disable',disable>} (fcn) is loc ne Ω then go to loc;;
        /*control functions:  wait, release, enable and disable are identified by the above 'if'.
        These functions do not require synchronization to avoid race conditions. The remain-
        ing requests, read, write, backspace, rewind, and space, are handled here.*/
reserve (file);
    /*The 'type' table classifies I/O devices by type. c.f. 3.5.1.1*/
if fcn ∈ {'read','write'} or n(type(iodevice) ∈ discs) then
        /*For disc read/write operations and for rewind, etc. operations on other devices (such
        as tapes) the physical I/O device on which a file is mounted must actually be reserved.
        On the other hand, for discs, control operations such as backspace and rewind are not
        physically carried out; the new position of the read-write head is simply recorded in
        system tables. Physical positioning takes place only when actually needed before a
        read or write request, since the read-write head may be repositioned due to activity for
        another file sharing this device. Thus, since no physical I/O will take place, the device
        need not be reserved. But for operations requiring the use of the physical device, we*/
    reserve(iodevice); /*reserve physical device*/
        /*A monitor call may result in a sequence of operations on a device (for example, a
        disc read requires a head-positioning operation, followed by a read operation), with the
        requirement that the device not be operated by other processes during this sequence of
        operations. Thus the system's I/O interrupt handler will not free iodevice when it
        completes an operation, since this will allow a competing process to sieze iodevice.
        Instead, we associate a blank atom with each device via the map rightouse, and reserve
        that blank atom, also. Each completion of a physical I/O operation will result in this
        atom being freed. Thus, the busy status of the blank atom can be used by the monitor
        services routine to synchronize itself with I/O operations. Only when a request is
        completed, will the monitor service process free iodevice, at which time iodevice
        becomes available for competing processes.*/
    reserve(rightouse(iodevice)); /*The above-mentioned blank atom.*/
    filehandledby(iodevice)=file;/*When an interrupt is received resulting from the comple-
        tion of the I/O operation on iodevice, the quantity filehandledby(iodevice) will be used
        by the I/O interrupt handling process to determine the program file for which the
        operation was performed. Because all processes reserve iodevice and
        rightouse(iodevice) in that order, and release them in reverse order, deadlock cannot
        occur [Hav], so long as I/O operations run to completion and cause an I/O inter-
        rupt.*/
    reserve(chan); /*get control of channel*/
end if;
go to {<'read',readwrite>,
    <'write',readwrite>,
    <'backspace',backspace>,
    <'rewind',rewind>,
```

```
        <'space',space>} (fcn);

readwrite: /*Treatmeant of 'read' and 'write' requests.*/
    if type(iodevice) ∈ discs then
            /*For discs, we must first position the read/write mechanism to where the user
            believes it to be. This is necessary since the disc drives may be shared by several
            processes, each of which causes the read/write mechanism to be positioned unknown
            to the other processes. It is the operating system's function to make this invisible to its
            users. Note that 'maxposition' and 'filebeginning' are macros introduced at the begin-
            ning of the monitor services section of code.*/
        position=logicalposition(file)+filebeginning;
        if logicalposition(file) ge maxposition then
                /*At this point, in a more advanced system, an additional extent could be allocated
                to the file. In that case, the computation of 'position' above becomes more compli-
                cated. However, we will merely signal an I/O error for this case.*/
            savecause(file)=badio; /*signal error if positioned*/
            free rightouse(iodevice); /*beyond area allocated to the file*/
            go to wrapupiorequest;  /*and return to caller.*/
        end if;
        seek(iodevice,position); /*physical operation to position the*/
        waitnotbusy(iodevice); /*read-write mechanism; wait for completion*/
    end if;  /*end of special processing for discs*/

            /*In a read or write request, the second element of parm gives the blank atom that is
            associated on the caller's workqueue with the data which is transmitted. See Section
            3.3.2.*/
    if fcn eq 'read' then
        read(iodevice,localbuffer); /*'hardware' read, see Section 2.3.2.*/
                /*'localbuffer' is a local area within the I/O service process, into which information
                will be read.*/
        putfirst(CPU,caller); /*Operation initiated -- allow caller to procede.*/
        waitnotbusy(iodevice); /*Delay monitor service process until operation*/
        putfirst(caller,<buffpart(parm),localbuffer>); /*is complete, then return data read.*/
    else
        findfirst(caller, x, hd x eq buffpart(parm));
        if x eq Ω then
                /*Buffer containing output data does not exist.  Error.*/
            savecause(file)=badio; /*Simulate error interrupt.*/
            free rightouse(file);
            go to wrapupiorequest;
        end if x;
        remove(caller,x);
        write(iodevice, x(2));
        putfirst(CPU, caller);
        waitnotbusy(iodevice);  /*Delay until operation is complete.*/
    end if fcn;
        /*Control returns here when physical I/O operation is complete.*/
    go to wrapupiorequest;

backspace: /*Treatment of 'backspace' requests.*/
    putfirst(CPU,caller);
            /*for discs, backspace is not physically carried out, instead, we just record where the
            user thinks he is positioned*/
        logicalposition(file)= 0 max (logicalposition(file) – 1);
        savecause(file)=goodio; /*simulate successful backspace*/
    else backspace(iodevice);
        waitnotbusy(iodevice);
    end if;
    go to wrapupiorequest;
```

```
rewind: /*Treatment of 'rewind' requests.*/
    (disable)
        putfirst(CPU,caller);
            /*For discs, we have only to rewind the file 'logically'.*/
        if type(iodevice) ϵ discs then
            logicalposition(file)=0;
        else
                /*For tapes, a physical rewind operation must be issued.*/
            rewind(iodevice);
            waitnotbusy(iodevice);
        end if;
    end disable;
    go to wrapupiorequest;

space:  /*Treatment of 'skip record' requests.*/
    (disable)
        putfirst(CPU,caller);
        if type(iodevice) ϵ discs then
            logicalposition(file)=(logicalposition(file)+1) min maxposition;
            savecause(file)=goodio;
        else
            space(iodevice);
            waitnotbusy(iodevice);
        end if;
        go to wrapupiorequest;
    end disable;

        /*Now, the entire monitor request is completed, and the iodevice may be free'd for use by
        other processes.*/
wrapupiorequest:  free iodevice;
transmitiointifany:
        /*This point in the code may be reached in two ways:
            1. As the result of a logical I/O operation terminating, or,
            2. As the result of a monitor-enable request to a file which has a stacked interrupt, in
            which case the interrupt gets unstacked (i.e. processed). Note that in this latter case
            the actual unstacking operation is done by 'enable'
            If the process 'caller', which owns the logical file 'file' has been in I/O-wait for the file
        'file', then the process is made dispatchable, and the interrupt for 'file' is processed. If the
        I/O-wait was for another file, then the interrupt for 'file' is stacked.
            Otherwise, if 'caller' is already in an I/O fixup, or if 'file' is disabled for interrupts, the
        interrupt is stacked. In all other cases, the interrupt is processed.
            In processing an interrupt, if 'file' had had a monitor-release request issued, the
        interrupt is ignored. Otherwise, the location counter for 'caller' is forced to 'file's fixup
        routine, and the main stream environment is saved if 'caller' was not in another I/O fixup
        at the time.
            In any case, the logical file 'file' is now available for further operations.

            Note in connection with the following code that user interrupt routines are not logically
        interruptable except in the case when a monitor-wait request has been issued in a fixup
        routine. In that case, if an interrupt had already been stacked for 'file', or, if 'file' is busy
        and has not had a monitor-release request issued for it, when the operation on 'file'
        terminates, the fixup routine for 'file' is executed as a co-routine to the fixup in which the
        wait request occured. Control does not return to the fixup routine which requested the
        wait.

            cf. section 3.5.1.5 for some of the maps which are used here.*/

(disable)
```

```
                                flow   (iowait(caller))?
                          (filewait(file))?          (disabled(file) or ioint(caller))?
    (iowait(caller)=false;)+      stack,     stack,              interrupt,
    (putfirst(CPU,caller);)+
    interrupt;
interrupt:
    if rel(file) then
            /*If a release request had been made for 'file', ignore the interrupt.*/
        rel(file)=false; /*Now we can forget that the release had been requested.*/
    else
                /*Interrupt the process which owns 'file'.*/
        (   if not ioint(caller) then
                    /*Process was in mainstate.  Save it for restoration on 'endfixup' request.*/
                mainstate(caller)=state(caller);
            end if not ioint(caller); )


            /*In any case, force location counter to fixup routine for 'file'.*/
        loctr(state(caller))=fixup(file);
    end if rel(file);
    filewait(file)=false; /*Wait is no longer pending.*/
        /*Save file i.d. and cause of the most recent I/O interrupt.*/
    interrupted(caller)=<file,savecause(file)>;
stack:
    if rel(file) then
        rel(file)=false;
    else
        putlast(<caller,io>,file);
    end if rel(file);
end flow;
free file; /*Monitor operation complete -- free program file.*/
term;
end disable;


wait: /*Treatment of 'wait' request.*/
    (disable) /*Find first stacked interrupt for the file, if any.*/
        findfirst(<caller,io>,x,x eq file);
        if x ne Ω then
            /*An interrupt is stacked for this file.  Force the user process to its I/O fixup routine
            for this file.*/
            remove(<caller,io>,file>;
                /*Note:  The 'wait' service is the only manner in which an interrupt routine may
                interrupt another interrupt routine.  In this case, the interrupt routines work as
                co-routines, and the second routine never returns control to the first.*/
            if n ioint(caller) then
                mainstate(caller)=state(caller);
            end if;
            loctr(state(caller))=fixup(file);
            interrupted(caller)=<file,savecause(file)>;
            free file;
            ioint(caller)=true;
            putfirst(CPU,caller);
            term;
        end if;
        if file ∈ busy then
            iowait(caller)=true; filewait(file)=true;
            /*The caller's process is not returned to the CPU's workqueue.  When the awaited I/O
            operation ends, then the caller will be returned to the CPU's workqueue.*/
            term;
        end if;
```

```
    endwait: putfirst(CPU,caller);
        term; /*Monitor service action for wait is completed.*/
    end disable;

giveinterruptcause:
        /*Return via the caller's workqueue, the status information associated with the most
        recently processed I/O interrupt belonging to this caller. In this sense, 'most recently
        processed' means either having entered a fixup routine, or having satisfied a wait condi-
        tion.*/
    putfirst(caller, <buffpart(parm), interrupted(caller)>);
    go to endwait;

enable: /*Treatment of 'enable file' request.*/
    disabled(file)=false; /*initialized in subroutine 'assign'/
    putfirst(CPU,caller);
    findfirst(<caller,io>,x,x eq file);
    if x ne Ω then
            /*Release the first interrupt which was stacked for 'file' while it was disabled.*/
        remove(<caller,io>,file);
        go to transmitiointifany;
    else /*No interrupts were stacked for the file.*/
        term;
    end if;

disable: /*Treatment of 'disable file' request.*/
    disabled(file)=true;
    putfirst(CPU,caller);
    term;

release: /*Treatment of 'release file' request.*/
    (disable)
            /*Previous interrupts for this file are forgotten, and if an operation is in progress, the
            interrupt resulting from the logical end of operation will be discarded without forcing
            control to the user's fixup routine.*/
        reports={x∈workset{<caller,io>} | x eq file};
        (∀x∈reports) remove(<caller,io>,x);;
        rel(file)=true;
        putfirst(CPU,caller);
        term;
    end disable;

endfixup: /*User program exits from I/O fixup routine.*/
    if not ioint(caller) then
            /*The caller was not in an I/O fixup routine. We elect to ignore the request.*/
        go to leave;
    end if;
    state(caller)=mainstate(caller);
    ioint(caller)=false;
    findfirst(<caller,io>,file, n disabled(file));
            /*Other program files await user interrupt processing. Take the enabled-file which
            completed it's operation first, and treat it as though a wait had just been issued for it.*/
    if file ne Ω then
        go to wait;
    end if;
        /*There are no interrupts stacked for enabled files.*/
leave:
    putfirst(CPU,caller); /*Caller may reume operation.*/
    term; /*Monitor service completed.*/
```

```
fixupaddr: /*Establish label of fixup routine associated with the file.*/
        /*For 'filepart' and 'labelpart', see the macro definitions on page 76.*/
    fixup(filepart(parm))=labelpart(parm);
    putfirst(CPU,caller);
    term;


                /*  ***************************  */
                /*        I/O INTERRUPT HANDLING       */
                /*  ***************************  */

    /*control reaches 'ioxpt' on I/O interrupts*/

ioxpt:
    (disable)

        /*First, do time accounting if a user program was interrupted.*/
      if resume ∈ userprogs then
        steptimeleft(resume)=timer - clock;
      end if;

        /*The interrupted process, which is independent of the just completed I/O operation,
        is returned to the head of the CPU's workqueue.*/

        /*Notes: 'devaddr' extracts the address of the device causing the I/O interrupt from
        cause. channelfree(cause) is true iff the interrupt signals that the channel has switched
        from the active to the free state. Similarly for devicefree and its argument.

        Only when a monitor routine (above) finishes interpreting an I/O request, does it cause
        control to go to 'wrapupiorequest', where the physical device is finally free'd, and made
        available to other processes. Here, it is also determined if the user should process the
        interrupt in a fixup routine.*/

      putfirst(CPU,resume);
      iodevice=devaddr(cause); /*extract address of interrupting device*/
      if channelfree(cause) then
        free channelpath(iodevice);; /*release channel if interrupt
                        indicates that channel became free*/
      if n devicefree(cause) then term;;
        /*Make the device available for other monitor services.*/
      free rightouse(iodevice);
        *See the comment on page 78 following 'reserve(iodevice);'. We don't free the device
        itself at this point in case it is being used by monitor serces for a sequence of operations
        which cannot be disturbed by other actions.*/
      savecause(filehandledby(iodevice))=cause;
      <CPUcontrol,loctr(state(thisprocess))>=<dispatcher, ioxpt>;
    end disable;
```

```
                /*  ************************  */
                /*          DISPATCHER        */
                /*  ************************  */

getwork: waitcopy=waitset; /* processes with awaited conditions*/
         /*First, we recheck all awaited conditions. cf. 2.4.3.1 for a discussion of the await
         verb.*/
loop: if waitcopy ne nl then
         s from waitcopy;
         L=loctr(state(s)); /*retest awaited condition*/
         CPUcontrol=s; /*control to process testing condition*/
         if isok then s out waitset;
            putlast(CPU,s);
         else loctr(state(s))=L;
         end if;
         go to loop;
      end if;
      /*Now we attempt to find a job to dispatch. Processes enqueued on 'CPU' are candidates.
      Non-user processes, i.e., system processes, are given priority.*/
      /* if waitcopy eq nl then */
   (disable)
      findfirst (CPU,s,not(s∈userjobs)); /*gives s as output*/
      if s ne Ω then
         remove (CPU,s);
         CPUcontrol=s;
      else if getfirst(CPU) is s ne Ω then
            /*The process just found must be a user-process, since our search for non-user
            processes had failed to find any, and, since we are disabled, no other process could
            have sneaked another process on the CPU workqueue. As a check on the software
            and the CPU, we could check that indeed, s∈userjobs, but we chose to omit this
            check for the meantime.*/


            /*Prepare to stop user if time estimate is overrun.*/
         timer=clock+steptimeleft(x);
         CPUcontrol=x;
      end if;
   end disable;
   go to getwork;



                /*  ************************  */
                /*    TIMER INTERRUPT HANDLER */
                /*  ************************  */

timexpt:
   (disable)
      if resume∈userprogs then
            /*A user's program has run out of time. Determine the process-id of the JCL
            interpreter corresponding to this user program, and notify it that the user has run
            out of time.*/
         interpreter=ancestor(initialvar(state(resume)));
         enqueue timeisup on interpreter for resume;
         steptimeleft(resume)=0; /*Indicate that time has run out.*/
      else
            /*A system program was interrupted. Set the timer to interrupt in one second, and
            restore the interrupted process to the head of the dispatcher's list.*/
```

```
            putfirst(CPU,resume);
            timer=clock+onesecond;
        end if;
            /*Give control to the dispatcher, and at the same time, set up the timer interrupt
            routine to start at 'timexpt' at the next timer interrupt.*/
        <CPUcontrol, loctr(state(thisprocess))>=<dispatcher, timexpt>;
    end disable;
end operatingsystem;
```

Chapter IV

A Multiprogramming System

4.1 System Objectives

The second operating system to be described is intended to run on a single CPU configuration operating in a non-interactive environment in a multiprogramming mode. 'Multiprogramming' will be taken to mean that more than one user run may be in progress at a given time. 'Being in progress' should be understood to mean that execution of a user job has commenced but not all the user processes associated with the job are concluded.

By adopting a multiprogramming design, one makes it possible to utilize a greater fraction of a computing system's resources than a single job is likely to require at any given time. This can give greater throughput from a computing system than is achievable by a uniprogramming design such as that described in Chapter III. (We observe at this point that even the simple uniprogramming system of Chapter III really utilizes multiprogramming in so far as several processes coexist in that system. However, at most one of these is a user process.) By keeping several processes available and allowing them to compete for resources we increase the likelihood of processes being enqueued on a resource; then when one process releases a resource, it can be utilized at once by another.

If it is unlikely that a user job requires as much as half of main memory, then there will generally be room for more than one user job to be resident in main memory at the same time. In this case, the objective of overlapping setup time with the execution of a non-setup job becomes relatively less important, since the multiprogramming activity of our system will probably keep one or more user jobs in progress during setup anyway. Thus, 'setup time overlapping' will not be provided explicitly in the next system to be described.

We shall wish to minimize the burden of change imposed upon a user who moves from uniprogramming to multiprogramming. In making this move, the user gets no increase in fuctional power; his gain lies in the greater availability of computing resources due to their better utilization.

In a multiprogramming system, symbolic referencing of I/O devices becomes mandatory. Several jobs are now to run concurrently, and the probability that two jobs can coexist will decrease markedly if jobs could demand to use a specific member of a set of functionally identical devices.

4.2 Multiprogramming Strategy Considerations

For multiprogramming to succeed in increasing system utilization, we must have programs concurrently in storage which are capable of using **diverse resources**. To take an extreme example, if all programs use only the CPU, multiprogramming cannot possibly reduce the total program running time. Contrariwise, if we are given a group of programs using several resources, then to the extent that a single program cannot concurrently use resources (e.g. CPU idle waiting for I/O

completion), idle resources become available to other processes. Recaptured critical resource idle time is a measure of the advantage gained by running a set of jobs in multiprogramming mode.

We want to enable our scheduler to select jobs for concurrent execution in a manner facilitating the overlapped use of several resources [C]. Suppose that with each job, we are given estimates of CPU time, disc transmission time, and tape transmission time. Then, for the total family of jobs in the system, we can compute the estimated workload for each of these resources. It is clear that the shortest estimated time to completion for the total collection of jobs is at least as great as the estimated workload on the busiest resource. (If there are several identical elements such as disc or tape channels, and assuming that the work can be evenly distributed among functionally identical elements, we divide the estimated load on the resource by the number of identical elements which can operate concurrently and which are available.)

Now consider some subset of jobs which can coexist on the computing system, that is to say, a subset whose total memory requirement does not excede the machine's memory capacity, whose total tape drive requirement does not excede the total number of drives, etc. For this subset, compute the estimated workload, which will be a triple. If the busiest resource of the subset is not the busiest of the entire workload, then the subset is out of balance with the total workload, and running it will surely increase the running time for the entire workload.

To aid in determining a good subset of jobs to run concurrently under multiprogramming, we wish to attach a measure m to a subset of jobs so as to indicate attractiveness of concurrent execution. That is, if P and Q are two sets of jobs, we would like m to have the property that if $m(P)<m(Q)$, then P is a better subset of jobs for multiprogramming than Q. We will call such a measure a *figure of merit*.

If we make the optimistic assumption that the heaviest loaded resource can be kept busy all the time by means of multiprogramming, we can estimate the fraction of time that a subset of jobs will keep each resource busy. For example, suppose that the estimated workloads for CPU, disc and tape are 10, 5, and 3 minutes respectively. Then the shortest time in which the subset can run is 10 minutes (if the CPU can be kept busy), and the CPU, disc channels and tape channels would be busy 100%, 50%, and 30% of the time.

To define out figure of merit, we compare these loadings, calculated for a job subset, with the loading calculated for the entire workload, and penalize a subset of jobs for each resource which (when the workload for the subset is normalized to make the load on the heaviest used resource equal to 1) is underloaded as compared to the total workload (normalized the same way). No credit is given for resources which are loaded more thoroughly in the subset than in the total job set. Thus, if $L=\langle l_1,l_2,l_3\rangle$ is the total workload of the full set of available jobs, and $ML=\mathbf{max}\{L\}$, if $P=\langle p_1,p_2,p_3\rangle$ is the total workload of a subset S of jobs, with $MP=\mathbf{max}\{P\}$, then as a figure of merit for the subset S we will take:

$$[+:1\leq i\leq 3]\ \mathbf{max}\ \{0,L(i)/ML-(P(i)/MP)\}$$

We consider the jobs in priority order, and we add to the collection of running jobs the oldest job of highest priority for which the figure of merit of the enlarged collection is not larger than the figure of merit for the current collection of running jobs.

4.3 Job Control Language

In keeping with our objective of imposing minimal transition difficulties on our users, the JCL of section 3.2 will be used with only minor changes in our multiprogramming system.

The TIME parameter on the job statement will be given as an n-tuple, where the first component is the estimated CPU time, and the remaining components are the estimated requirements of the other system resources.

A SPACE parameter will be included on the job statement to specify the amount of contiguous main memory space which the job requires.

4.4 Organization

The overall structure of our multiprogramming system will be similar to that of the uniprogramming system of Chapter III. That system already employed multiprogramming in order to run several operating system processes concurrently with at most one user program. Thus, in order to multiprogram among several user jobs, we expect that the main changes will be to the scheduler, so that it can provide several user programs for concurrent execution.

The input reader must be modified to accept a more comprehensive time estimate, and to update the system's running estimate of the backlogged workload. Only minimal change to the handling of job cards is required. The output writer is unaffected by multiprogramming considerations.

The scheduler is the component which is the most affected by multiprogramming. It must recompute the system's workload and figure of merit whenever a new user job is encountered or a user job terminates. It must then determine if jobs exist which can run with only the system's unused resources and which would also not degrade the system's figure of merit. If such a job exists, it is sent to job control for execution.

The dispatcher must also be reworked to account for multiprogramming. First, system processes, all of which are known to use CPU bursts of short duration, will be given priority over non-system processes. If there are no system processes in the CPU's workqueue, we will choose the oldest user process. There are many alternative dispatching algorithms depending on the operating system's objectives. The dispatcher we will use tries to get jobs completed in FIFO order.

Among the factors which must be assessed in determining whether a program can be added to a set of running programs is the amount of main storage which it requires. The management of main memory can be a complex matter, and it is strongly dependent on the addressing characteristics of the machine. We will assume that the job statement specifies the amount of contiguous space which the job requires, and that object programs are "relocatable", that is, that a program can run

in any portion of main memory provided that sufficient contiguous space is available. We will also suppose that a job holds onto a fixed amount of memory from start to finish. Of course this is by no means the only strategy available. We could elect to swap programs between main memory and disc storage, or to move programs from one part of main memory to another to create large blocks of contiguous storage, provided that the machine architecture allows programs to be relocated after loading and partial execution.

In this system, we assume that the machine architecture does not allow relocation of a program after partial execution. The system described in Chapter V does assume that dynamic relocation is possible. It would be a minor matter to incorporate the code at label 'getblock' of section 5.7 in the scheduler, which is responsible for memory allocation in this system.

To allocate resources one must partition main memory and I/O devices among competing jobs. Deadlocks can arise when several programs are allocated resources from a common pool. Deadlock occurs when two or more processes are waiting for resources held by the others. Neither can until it gets the additional resources which it needs. There are several ways to avoid this situation, but one must be aware of the ways in which deadlock can occur in order to avoid it. In our system, had we left resource allocation to the job control analyzer, we would have faced the possibility of deadlock, since several job control analyzer processes can be in operation concurrently. Instead, this function was reserved to the scheduler, of which there is only one instance in the system.

Besides allocating a portion of available main memory to a job, our system must insure that the job is confined to its allocated memory. A component of the environment will be used to specify the memory areas available to a process. The boundaries will be given as a set of triples of the form <m,n,x> to indicated that n contiguous words of memory beginning at m are accessable to the process. An attempt to violate boundary restrictions results in a memory-protect interrupt. x specifies the types of accesses which the process may make to the memory area. We consider any combination of the following types of accesses: execution, read-data, and write-data.

If our programs were carefully constructed by the compiler to keep data areas separate from executable code, then the loader can specify user boundaries to make the executable code have only "execute" access, user defined constants to have only "read-data" access, and user defined variables to have both "read-data" and "write-data" access. Such a mechanism makes it impossible to execute data due to a wild branch, or to accidentally modify a program or constant due to faulty indexing, or to modify a constant by using it as an argument to a subroutine with side effects. An attempt to violate boundary restrictions results in a memory-protect interrupt.

When the loader sets up the user program, the user space which the loader uses as a data area has both read-data and write-data access. The loader then can partition the user space into execute only, read only, and read-write areas before the job step executes. Of course, this implies that the programs being loaded carry sufficient information to allow the loader to determine the partitioning.

We can now fill in a gap in Section 3.4 regarding the ring structure of our operating system. Innermost ring processes, the interrupt handlers, start with read-write-execute access to all main memory. It is conceivable that such processes restrict their own access as the nature of their task

develops. Thus, while the monitor interrupt handler and I/O interrupt handlers have maximum access privileges, the processes which they invoke should have limited memory access. Allocation routines, scheduler and job control do not require write access to the special sets described in Section 2.4.1, with the exception of **workset**, nor execution access to the interrupt routines.

Since so much of the uniprogrammed system carries over to the multiprogramming case, only those routines which are extensively modified will be presented in their entirety below. For routines having only minor modifications, only the modifications will be given.

```
/*  ***************************  */
/*          INPUT SPOOL          */
/*  ***************************  */

/* The only changes to input spool involve the new SPACE parameter and the modified TIME
   parameter, which is now a triple, giving CPU, disc, and tape time estimates. Refer to input
   spool routine for uniprogramming, page 56.

   For each job j, esttime(j) is a triple giving the estimated usage for CPU, disc, and tape. The
   tuple estwork is maintained by the operating system to be the vector sum of the elements of
   esttime, so that at all times,
         estwork = [+,1≤i≤3]<[+,v(j)∈esttime]v> */

      .
      .
      .
jobcard: if jobcardprocessed then go to endcard;;
      /*The next job card encountered will mean end of current job.*/
   jobcardprocessed=true;
   (∀y∈{'time','priority','space'})
      if image(y) eq Ω then
         image(y)=profile(username,y);
      end if;
   end ∀y;
   go to mainloop;
      .
      .
      .
endcard:
      .
      .
      .
      /*No JCL errors found with job just read. In the multiprogramming version, we also must
      adjust the workload estimate when a JCL error free job has been read. If the time parame-
      ter is not a 3-tuple, we use a default time-estimate vector for it.*/
   job in movers;
   owner(job)=username; /*Setup map showing job's owner.*/
   esttime(job)=if type tempjobcard('time') ne tuple
                     or #tempjobcard('time') ne 3
                     or ∃est(i)∈tempjobcard('time') | type est ne integer then
                     profile(username,'time')
                  else tempjobcard('time');
   if type tempjobcard('space') ne integer then
         /*Improper space specification, use profile.*/
      tempjobcard('space')=profile(username,'space');
   end if type;
   if type tempjobcard('priority') ne integer then
         /*Improper priority specification, use profile.*/
      tempjobcard('priority')=profile(username,'priority');
   end if type;
   space(job)=tempjobcard('space'); /*User's estimate of maximum space required.*/
   priority(job)=tempjobcard('priority') min (maxprio(username));
   resources(job)=volumesneeded; /*volumes needed by job into system tables.*/
   newallocations(job)=newfiles; /*Newly catalogued entries due to this job.*/
   close(expandedjcl,'jclopenclosename');
(disable)
   (∀y(i)∈esttime(job)) /*update total workload estimate*/
      estwork(i)=estwork(i)+y;
```

```
        /*A corresponding decrease of estwork is made when a job terminates. See the
        ∀-block following the label 'endjob', below.*/
end disable;
    enqueue <job,expandedjcl> on scheduler for username;
    if image('command') eq 'job' then go to newname;
    else go to discard;;



                /*  ************************  */
                /*        SCHEDULER           */
                /*  ************************  */

    /* The scheduler is enqueued when:
        a) input spool has read a new job, and when
        b) job control terminates a user job
        Replaces scheduler, page 66.*/

schedule:  await #workset{thisprocess} gt 0;
    c=getfirst(thisprocess);
    if ancestor(c) ∈ users then /*true for case a*/
        if (∃x(i)∈backlog | priority(x(1)) lt priority(info(c))) then
                /*insert new item after all jobs having the same or higher priority but ahead of jobs
                having lower priority*/
            backlog(i:)=info(c)+backlog(i:);
        else backlog(#backlog+1)=info(c);
        end if ∃x(i);
    end if ancestor(c);

        /*Note that in case b, for which no explicit code is provided in what follows, we put an
        item on the scheduler workqueue to 'awaken' it. After 'awakening', it may find that
        additional work can be scheduled, since the termination of an old job will have freed some
        resources.*/
newmerit:
    if #backlog eq 0 then
            /*There are no jobs to consider for scheduling.*/
        go to schedule;
    end if;
    m = merit(workload);
        /*If no resources are underloaded and there are at least two user programs running
        concurrently, the scheduler waits to have another request enqueued on it.*/
    if m eq 0 and #userprogs gt 1 then
        go to schedule;
    end if;
        /*find first job for which there are enough resources and which does not increase the
        figure of merit*/
    (∀x(i)∈backlog)
        <job,input>=x;
            /*Set 'testmerit' to the figure of merit of the set of currently scheduled jobs augmented
            by the candidate job, 'job'. If the figure of merit does not improve, i.e., diminish, then
            consider a different candidate.*/
        testmerit=merit([+:z(j)∈esttime(job)]<z+workload(j)> is testworkload);
        if testmerit ge m then continue ∀x(i);;
        (disable) /*to preserve integrity of space tables*/
                /*For a potential job to be run, first determine whether there is sufficient main
                storage.*/
            if not(∃y(k)∈freesize | y ge space(job)) then
                continue ∀x(i);
            end if;
```

```
            /*Now form the set of devices needed, and for each device, the set of files for this
              job on that device.*/
          newdevices=nl;
          (∀vol∈resources(job))
              if (∀device∈devices*ready | mounted(device) ne vol) then
                    /*Volume must be mounted.*/
                  dev=arb{d∈devices | type(d) eq catalogue(vol)('device')
                              and mounted(d) eq nl};
                  if dev ne Ω then /*job might be selected*/
                      <dev,vol> in newdevices;
                  else /*Not enough devices for this job now.*/
                      continue ∀x(i); /*Look at the next job.*/
                  end if dev;
              end if (∀device;
          end ∀vol;
              /*Sufficient resources, and an improvement of the figure of merit. Prepare to
              interpret JCL. First, update the tables to account for the main memory to be used
              by the selected job.*/
          <origin,freesize(k),freeloc(k)>=
              <freeloc(k),freesize(k)-space(job),freeloc(k)+space(job)>;
          if freesize(k) eq 0 then
              freesize(k:)=freesize(k+1:);
              freeloc(k:)=freeloc(k+1:);
          end if;

              /*Create output file for the new job.*/
          catalogue(<job,newat> is output) =
              {<'device',tempdisc>,<'space',filesize>, <'disp',{'cat','leave'}>};
          allocate (output);
          userprogs(job)=newat;
              /*Create a job control analyzer process.*/
          split to <<job,newat>, jobmonitor> <input,output,origin> for thisprocess;
          workload=testworkload;
              /*Issue mounting messages for volumes required by the selected job.*/
          (∀vols∈newdevices)
              mounted(hd vols is dev) = newdevices(dev);
              operatormessage('mount',mounted(dev),'on device',dev,'for user',user(job));
          end ∀vols;
          go to newmerit; /*one job found -- now look for more*/
      end disable;
  end ∀x(i);
      /*No more jobs can be scheduled at present.*/
  go to schedule;
  end scheduler;



          /*  ************************  */
          /*       JOB CONTROL INTERPRETER       */
          /*  ************************  */

define jobcontrol;
      /*Job control now has an additional parameter transmitted through initialvar, the origin of
      the memory space assigned to the job. The size of the memory space is in the system table
      called 'space'.*/

jobmonitor:
  <input,output,origin>=processparameter;
  timeleft(mov)=esttime(mov)(1);
```

```
endjob: close (hd output,outputname); /*User's output file is complete.*/
    unhook(outputname); /*Detach it from job control interpreter.*/
    enqueue outfile on outspool for mov; /*Pass output to output spool.*/
    unhook(inputname);
    userprocess out userprogs;
    enqueue nl on scheduler for thisprocess; /*Reawaken the scheduler!*/
    resources(mov)=Ω; /*Allow job's volumes to be released.*/
(disable)
        /*Update the estimated workload*/
    (∀y∈esttime(job))
        estwork(i)=estwork(i)–y; /*Overall workload*/
        workload(i)=workload(i)–y; /*Multiprogramming-set workload*/
    end ∀y(i);
    esttime(job)=Ω;
        /*Update the system table of available memory space by marking the space of the terminated job as free. If two or three blocks of free memory are contiguous, combine them into one large free block.*/
    i=[max:x(k)∈freeloc | x lt origin] k
    if i eq Ω then
        if origin+space(mov) eq freeloc(1) then
            freesize(1)=freesize(1)+space(mov);
        else
            freesize=<space(mov)>+freesize;
            freeloc=<origin>+freeloc;
        end if;
    else if freeloc(i)+freesize(i)= origin then
        freesize(i)=freesize(i)+space(mov);
        if freeloc(i)+freespace(i) eq freeloc(i+1) then
            freesize(i+1:)=freesize(i+2:);
            freeloc(i+1:)=freeloc(i+2:);
        end if;
    else
        freeloc(i:)=<origin>+freeloc(i:);
        freesize(i:)=<space(mov)>+freesize(i:);
    end if;
    term;
end disable;
    .
    .
    .

                /*  ***************************  */
                /*        FILE MAP MAINTENANCE   */
                /*  ***************************  */

    .
    .
    .
define unhook(programfilename);
    .
    .
    .
        /*Indicate that the unhooked file is no longer active on the physical device on which it resides. If no program files are assigned to the device, indicate that the device does not have a volume logically mounted. However, volumes required by the scheduler as indicated by resource(job), are unaffected.*/
    if ∀pfn∈programfiles | catalogue(userfile(pfn))('volume') ne mounted(d)
            and not(mounted(d)∈resources[moverpart{userjobs}]) then
```

```
.
.
.

/*  ***************************  */
/*          FIGURE OF MERIT      */
/*  ***************************  */

   /*See the discussion of figure of merit in section 4.2.*/
definef merit(v);
      /*em is the estimated workload of the heaviest loaded resource in the total workload of the
      system.*/
   em=[max:x(i) ∈ estwork]x;
      /*vm is the estimated workload of the heaviest loaded resource in the subset of jobs whose
      load-vector is given by v.*/
   vm=[max:x(i) ∈ v]x;
   return [+:x(i) ∈ v](0 max (x/vm-(estwork(i)/em)));
   end merit;

.
.
.
```

4.6 Possible Enhancements to the Batch Systems

A variety of enhancements to the systems presented in Chapters III and IV can be made with reasonable effort and without changing the systems' overall structure.

The input spool reader assumes that JCL is presented in standard SETL format (see Appendix A). This assumption is made for pedagogical reasons to avoid cluttering the code with parsing routines which are only incidental to the system's structure. To accept conventional input, the SETL **read** statements at 'mainloop' and 'discard' in input spool should be replaced by the SETL **record** statement to get the JCL statement as a string of characters, followed by a call to a parsing routine which converts the string to SETL structures. A more versatile JCL macro system which allows additional parameters to be passed appears in the interactive system of Chapter V.

The set of operator commands is quite limited. Rather than just allowing a 'hi priority' command to advance a selected job to the head of the scheduler's queue, we can allow the operator to request the following:

a) Reassign priorities to jobs in the scheduler's queue,
b) Terminate running jobs,
c) Remove jobs before beginning execution,
d) Prevent printing the output of a completed job,
e) Terminate the printing of undesirable output.

Enhancement a) requires that we overwrite priority(j) for job j based on the operator's input. To be effective, the scheduler must also be modified to take the priority data structure (which is already being maintained by the system) into account.

Modification b) requires that we determine the process-id of the job which the operator wishes to terminate, and simulate a "monitor('abend');" statement. Enhancements c) and d) merely involve examining the workqueues for the scheduler and output spool, and purging the appropriate item. In case c), we also require some bookkeeping operations which purge certain system data-structures of items related to the cancelled job.

It is more difficult to implement a command allowing the operator to suspend a running job until a later time, while allowing other jobs to run. Device-positioning must be saved in case other volumes are mounted on a suspended job's I/O units, and, if our multiprogramming system runs on hardware which does not support moving executing programs from one place in memory to another, then the job will have to occupy the same memory locations when it is resumed.

The operator can also play a greater role in describing the status of the various hardware components to the operating system. By declaring certain components as being unavailable, the operating system could continue in a degraded mode in which jobs requiring the unavailable devices would be delayed, while other jobs could be run. We could also allow the operator to override the system's device allocation given in mounting messages, although this can be tricky, since the system amy have made additional allocation decisions based on the one which the operator has overridden.

The device allocation features of our batch systems can be strengthened in several ways. First, a better choice of device on which to mount a volume can be made by trying to balance the load on all channels having identical devices. This is a relatively easy modification, requiring use of the 'channel', 'units', and 'estwork' data structures. Alternatively, the operating system need not dictate to the operator which of several identical units to use for volume mounting. The operator can indicate his choice of unit via a command, or, if the hardware generates a distinguished interrupt at the conclusion of mounting, the software can deduce the device chosen without further interaction by the operator.

Another allocation problem arises when a data file being written becomes larger than the declared file size. The 'readwrite' section of the monitor services routine detects this condition when the logical file position is compared to the maximum file size. At this point, the 'allocate' routine would have to be invoked. Some care in design is needed to avoid endless recursion, since 'allocate' uses I/O. The 'allocate' routine should probably give preference to free space on the volume(s) on which the file already has its extent(s). The catalogue's structure already permits multiple extents for a file (i.e. non-contiguous blocks), but our systems do not exploit this.

Several alternatives exist to the scheduling and dispatching algorithms used in the systems which have been presented. In our sample code, all scheduling decisions are reserved to the scheduler, with very minor interaction from the JCL interpreter in the uniprogramming system to achieve overlap of setup time. Hence, experiments with schedulers can probably be carried out with no structural change. Dispatching decisions are almost entirely confined to the dispatcher. In some instances, after a monitor service routine completes an activity, it does reactivate its caller by placing it directly on the CPU workqueue. In experimenting with dispatchers, some care regarding such modifications of the CPU workqueue may be needed.

More difficult, pervasive changes are required to add consistency checks to the operating systems. Such checks are almost totally absent in the code presented here, although these checks are the only software analogue to hardware error detection circuitry. As an example, the rereading of JCL by the JCL interpreter has no error checks, because it operates on the assumption that input spool would have rejected the job if it contained JCL errors. This assumption ignores possible I/O errors on the one hand, and possible processing errors on the other. There are also many instances in the I/O interpretation part of our code where the system data structures are assumed to always correctly represent the state of the I/O devices, although this can be checked by interrogating the I/O devices' status.

System support functions are lacking. A bootstrap loader is required to load the first portion of the operating system when the console load-button is pushed at the start of operations. Because of the extreme machine dependence of such programs, no dictions have geen put into PSETL to describe the necessary actions.

In any actual operating system it will from time to time be useful to reclaim unused space on shared secondary storage volumes. That is, the files should be rearranged so that all the unused space is contiguous. A program to carry out this function can be written to run under the operating system, provided it can be given special privilege to access the system's catalogue. It can be implemented most easily as a 'stand-alone' program, that is, a program not multiprogrammed with

other activity. However, it can also run multiprogrammed, but it must then interact with the scheduling and allocation routines to avoid disrupting files already in use.

Our systems have been written as though all the system's data structures were resident in main memory. Often this is impractical, and the operating system must resort to I/O to access these structures. With some care, our implicit assumption about main memory residence can be removed without major modification of the code. References to non-resident data structures become subprogram calls, and for each such structure a subprogram is required to manage it. If this approach is adopted, then some care will be required to conform to our restrictions on the use of disabled blocks.

In reality, computing systems have finite storage resources. In recognition of this limitation, users cannot be allowed to specify arbitrarily many files of arbitrary size which are to be catalogued by the system. A limit should be set on the total storage which a user's files may occupy; this limit should be enforced by 'allocate'. Similarly, a limit should be placed on the number of files which a user can enter into the system catalogue, for to do otherwise would require the catalogue to grow to unmanageable proportions, and the time to access information in the catalogue would become excessively long. Among the methods which can be employed to determine each user's limits are: divide the available resources equally among all users, require the system administrator to specify maximum allocations for each user, assign additional space to users who access a high percentage of their files, and require owners of inactive files to copy them from shared volumes to private tape and to drop the file from the catalogue.

Chapter V

An Interactive System

5.1 Introduction

Our previous operating system examples were oriented to batch processing, in the sense that the systems were not designed to allow the user to interact with his running program. We shall now describe an interactive system, which allows the user to remain an active participant during the computer's execution of the job.

There are several factors which motivate allowing users "on-line". First, it is often difficult to anticipate the outcome of a job step, and even more difficult to anticipate the correct remedial action in the event of a job step's failure. The JCL mechanism provided in the preceeding systems allow job steps to be sequenced in the absence of the user, but JCL is clearly not a powerful language. Giving JCL greater flexibility would increase the problem of debugging JCL code. This difficulty can be resolved by allowing the user to sit at a console and determine the appropriate actions as various situations arise.

Moreover, even when a program is debugged, the data we wish to give it may depend on the results of previous job steps. The user likes to be in a position to see partial results and enter new data.

Using terminals, several interactive users can share a computer by the use of multiprogramming, but some radical adjustments have to be made to the systems of Chapters III and IV before we have a reasonable time sharing system. Two of our multiprogramming strategies have to be completely replaced: we do not only wish to permit a subset of jobs to run concurrently if the total resource requirements of the subset does not exceed machine capacity, and we may have other aims in scheduling than simply to optimize a performance figure of merit. With users on-line, the delays associated with batch processing are unacceptable for certain interactions which are known to take very little computation. Thus, requests recently received should have relatively high priority for scheduling. Conceivably, every new request might get dispatched almost at once, but since this would overcommit main memory and CPU resources, some form of preemption must be possible, and substitute memory must be used for a preempted job. The dispatcher must adjust its priorities; a job which uses substantial resources is clearly not one for which quick response should be expected, and such a job's priority may decrease to make possible quick response to "trivial requests".

Note that, as is now conventional, we view the system's storage as a hierarchy of memory devices. The fastest, smallest of these devices is main memory. Then, in order of decreasing performance and cost per bit but increasing capacity, are devices such as slow bulk core storage, drum, disk and tape. These devices will be called secondary storage. Only main memory (and in some cases, bulk core storage) has the property of being directly accessible by the CPU. If the total memory requirement of the scheduled jobs exceeds main memory capacity, some of the main memory images must be kept on the slower devices in the form of files. Then, a main memory image must be copied from secondary storage to main memory before the corresponding user can execute his program. Likewise, it may be necessary to move a program from main memory to secondary storage to make room in main memory for another user's program. The strategy for memory

management depends on many factors: performance requirements, limitations on the fraction of primary storage available to any one user, and the hardware addressing scheme.

Device Type	Access Time	Capacity (bits/device)
Bulk core storage	1 to 10 μs.	10^6 to 10^8
Drum	2 to 30 ms.	10^7 to 10^8
Disk	30-300 ms.	10^7 to 10^9
Tape	7 ms-1 min.	10^7 to 10^9

Table 5.1 Typical Memory Hierarchy Device Characteristics

5.2 Command Language

Since the user is interacting with the system, there is less need for him to put together a sequence of commands to be performed without further user action. Commands will therefore be treated as imperatives, to be executed immediately. The command language will be significantly richer than that defined for our batch system; we will introduce a more general command structure which allows each user to also introduce his own commands.

A command will have the form:

<cname,a_1,a_2,...,a_n>

where cname is the command's name (the verb of an imperative, so to speak), and a_1,a_2,...,a_n are arguments. Corresponding to 'cname', there must exist a file in a library, which consists either of an executable program, or of a sequence of more primitive commands. In the first case, the program is executed and passed the parameters. In the second case, each command in turn gets executed with the arguments substituted for parameters in the library commands. We will also provide for commands to be invoked from running programs by means of monitor calls. The command shown above can be invoked from a running program as follows:

monitor (cname, a_1, a_2, ..., a_n);

With this facility the user can be spared the trouble of issuing standard commands; an auxiliary program can do this for him.

In the system to be described, the code needed for carrying out the various commands is not generally kept in main memory. Command interpretation makes use of a loader which moves the appropriate code to carry out the command to an available portion of main memory.

The interactive system to be described avoids many of the JCL-related complications which arose in connection with the previous systems, because commands are analyzed only once. In the batch system, JCL is first analyzed by 'input spool', which collects information about a job into system tables. The scheduler uses these tables to make its scheduling decisions, and the job control

interpreter checks these statements again, perhaps bypassing conditional job steps, and also specifying the devices on which certain files should be mounted.

In the interactive system, commands are executed immediately after analysis. A reference to a dismounted file will cause an operator message to be issued, and the user will experience a delay until the mounting can take place. However, in a well designed time-sharing system, the vast majority of active files are always kept on-line. To make this possible, the operating system must include code which allocates on-line secondary storage among users and which causes inactive files to be automatically moved to archives which either are on-line but have very long access times, or which are off-line. (This aspect of storage hierarchy management is not shown in the code which follows.)

Having said all this, we now go on to list and describe the most significant 'primitive' commands, all of which we assume to be provided with the system.

5.2.1 Logon

The **logon** command has the form:

<'logon',user-name>

where user-name is the string of characters by which the user identifies himself to the system. The logon command must be the first command issued in a user session. (A "session" is the interactive analogue to the batch system's "run".) If the logon-program finds that the user's name is in its set of valid user names, it will ask the user to produce a password, i.e. a second identifier known to the user, but presumably not to any other user of the time sharing system. The password provides an extra measure of security for the interactive user against unauthorized use of his files or his resource budget.

We point out that a password as a means of identification is necessary, because it will be inconvenient to have users keep their user-names secret. It will be advantageous to have users communicate among themselves, and the user-names will be the means by which they identify themselves and the other users with whom they wish to communicate.

If the password given is correct, the logon-program executes the command 'setup'. This allows the user to automatically have certain actions performed on his behalf whenever he starts a session, rather than to have him reissue commands manually for these actions. Of course, it is the user's responsibility to define a file called 'setup'. If none exists, then no additional action is taken by the logon-program. After 'setup' is executed, the system awaits a command from the user.

5.2.2 Logoff

The **logoff** command has the simple form:

<'logoff'>

It is used to end a session, and is a signal to the system that memory space and CPU time will no longer be needed by the user who logged-off. In addition, if there is a hierarchy of devices on which files are stored, the user's files may be moved to a slower level of the hierarchy.

5.3 Main Memory Structure

If we were to assume that the programs for all active users fit into main memory, then certain very significant problems of memory allocation and performance which arise in connection with interactive systems would be hidden. We therefore choose to represent 'memory management' issues explicitly, and to assume that the total user demand for main memory exceeds the supply.

The memory management scheme chosen for use in connection with any particular timesharing system will always depend on the nature of the hardware. For simplicity in the treatment to follow, assume that our hardware has a base and bound relocation mechanism, in which a base B and a bound L are specified. A reference to main memory location x will be treated by the hardware as a reference to x+B, provided that $x<L$. An error interrupt results if $x \geq L$.

We will assume that there is a component of the environment called **relocate** which contains the appropriate pair <B,L> for the process. The code for a running process p may be moved from one contiguous area of memory to another, so long as the first component of relocate(**environment**(p)) is adjusted accordingly. Such a relocation scheme makes it possible to force all running processes to one end of main memory in order to leave a contiguous block of unused memory of maximum size.

Long running processes may be preempted in order to permit newly started commands to be executed (see 5.1). Such processes may have their main storage contents copied to a slower level of the memory hierarchy until the scheduler tries to resume execution of the process, at which time the memory contents are copied back to an appropriately large, available, block of main memory, but not necessarily the same area it had occupied previously.

When the operating system exploits the memory relocation feature, it will move user areas within main memory, or between main memory and secondary storage. To show this code, we need a diction which allows us to reference specific locations in the main memory. We will use a special variable, **memory**, to represent all of main memory as a vector. Consecutive components in the vector are contiguous locations in main memory. References to **memory** are not relocated; thus the operating system can control memory even if its relocation base is non-zero. Processes which do not have system privilege cannot access **memory**.

5.4 The File System

Users, communicating with the computing system from a terminal, possibly at a remote site, will prefer to keep their data in the system's memory hierarchy between sessions. This data includes not only information to be processed by the user's programs, but also his programs, both in source and object forms, partial results for future computations, and files of commands. We will refer to this collection of information as the 'file system'.

The file system is the most important component of a time sharing system. With its loss, a user community would have to reconstruct all its data and programs to become operational. With the file system intact and a loss of the CPU, a substitute CPU would enable the user community to work; since the users are remote from the CPU, the substitute CPU need not even physically replace the original one. If all user programs are in a higher level language in which underlying

machine details are suppressed, a user community could operate on several dissimilar CPUs, provided only that compilers for the higher level languages are available for the various types of CPUs.

5.4.1 File Names

We allow file names to be simple names or compound names, where the components of the name are separated by periods, e.g. 'source.squareroot'. Compound names can be used to differentiate in a consistant manner between several files relating to the same object. For example, associated with a square root routine may be the source program in SETL, the object program in a form suitable for use by a program loader, and a listing of the program as produced by the SETL compiler, showing the translation produced as well as diagnostics. In this example, we've identified three different files associated with the square root routine, and these might be named source.squareroot, text.squareroot, and listing.squareroot, respectively. Then, in order to compile, one might issue a command such as:

<'compile', 'squareroot'>

and the compiler, itself, will issue a file command for source.squareroot to obtain its input, and text.squareroot and listing.squareroot for its machine language and printed outputs. The user, in communicating with the loader, need just reference 'squareroot' again, because the loader, by convention, accesses only files with compound names beginning with 'text'.

5.4.2 Commands for the File System

Files, especially those kept in the memory hierarchy previously described (as opposed to being stored externally) are the most important objects in the time sharing system. All of a user's information is kept in files from session to session. As before, we must distinguish between user file names and program file names. A **file** command is provided to give a user-name to a file. The most common form is:

<'file',f>

Such a specification defines a file with user file name f to be in the storage hierarchy and sufficient space will be found for the file within the user's limits for his total storage allocation. No further parameters are ordinarily necessary although the other parameters of section 3.2.3 are still available for defining files not in the storage hierarchy which is managed automatically by the system. Files remain in the storage hierarchy from session to session, unless removed by the command:

<'erase',f>

The relation between a program file name and a user file name is established by the **define** command:

<'define',p,f>

In the above example, the program file name p is associated with user file f. The relation between a program file name and a user file name holds only throughout one session or until the program file name is redefined during the session.

Since commands may be executed by running programs, many programs will be designed to issue file, erase, and define commands and relieve the user from explicitly giving such commands from the terminal.

5.4.3 Communication between Users

In our previous discussions of operating systems, there had been a strong emphasis on the separation of users. The **boundaries** portion of an environment (cf. section 4.4) was devised to limit a user's reference to his own portion of main memory; the relocation mechanism (Sec. 5.2) likewise restricts a user to his allocated memory. The file naming scheme of section 3.2.3 and the supporting table structure of 3.5.1 prevent a user from accessing files other than his own.

With multiple users on-line, the use of the computing system as a communication medium becomes feasible. First, we can simply allow messages to be passed from one user to another by a simple command such as:

<'message',u1,'text'>

This would result in

'From u2: text'

printing on u1's terminal, where u2 is the user-name of the user who had given the 'message' command.

A more interesting aspect of communication is the sharing of files. We provide a command by which a user may specify that a certain file may be used by other users:

<'permit',f,s,c>

where f is a user file name, s is a set of user names, and c is a set of codes designating the types of access which may be made to the file, e.g. read, write, execute, or erase. In the case of execution access, the file is available only to the system loader when used by a sharer. That is to say, read or write requests will be rejected if issued by a sharer's process.

A user u2 who is permitted access to a file belonging to u1 indicates the name he gives to the shared file by a command such as:

<'share',ufn2,u1,ufn1>

where ufn2 is the user file name by which user u2 will refer to the file, and ufn1 is the name which the owner u1 had given to the file. The effect of **permit** and **share** commands remain in force across user sessions, until permission to share is withdrawn by means of a **permit** command specifying no access codes, or by dropping the user file name of a shared file from the catalogue.

We also allow a user who is sharing a file to permit others to share the file, providing that he only allows those types of access which he himself had been permitted to use.

To support file-sharing we must use a more complex catalogue than would otherwise be necessary. For each file f, catalogue(f)('permit') is a set of pairs of the form <u,x>, where u is a user (other than the owner), and r is the set of access rights which user u has to file f. For an unshared file, catalogue(f)('permit') is **nl**. If a file g results from sharing user v's file h, then catalogue(g)('share') **eq** <v,h>, whereas if g does not result from sharing, catalogue(g)('share') **eq** Ω.

5.4.4 Libraries

Related data files may be organized into a structure which we will call a library. A library is a file whose contents is a list of file names. This structure facilitates transmission of a set of files to a subprogram or to another process. Libraries may be shared between users in the same way as ordinary files are. When a user is allowed to share a library, all the files in its index become accessible to the sharer.

The operating system itself consists of several libraries, some of which only it may access, and some of which are accessible to users. The subprograms which interpret the various operating system commands are kept in a library of relocatable programs. Programs of use to the general user community are kept in system libraries to which all users have read and execute access privileges. A user who develops a package of routines which he feels is useful to others can create a library and permit similarly general use of it.

Certain system routines will search sequences of libraries for particular files. Examples of such routines are those involved in command interpretation, program loading (subroutine libraries), and compilation (macro libraries). Associated with each such routine and each user is a vector of library names. Initially, the vector may contain only the name of the system supplied library. By using a 'searchorder' command, the user can replace the vector with one of his own, thereby causing his libraries to be searched. To avoid his having to remember the name of the system's library, an '*' is used instead. Thus, if a user wants two of his libraries to be searched in program loading, one before the system library and one after it, he would give the command:

<'searchorder','loader','lib1','*','lib2'>

The effect of a searchorder command remains in force until another searchorder command is issued for the same function, or until the end of the current session, whichever comes first. If a user wishes to establish certain library search orders as a matter of course, then corresponding commands should be made part of that users 'setup' file (see 5.2.1), so that they will be issued automatically at the start of each session.

5.5 Organization

Our system will be divided into three major portions: a nucleus, privileged system commands, and non-privileged system commands. The nucleus is always present in main memory, has a relocation base of 0 and access to all main memory. Privileged commands and non-privileged commands have a non-zero relocation base and thus cannot directly access all the system data, but special

monitor services enable them to access part of this data. Non-privileged commands are indistinguishable from user programs.

5.5.1 The Nucleus

The nucleus consists of interrupt handling programs which are largely concerned with scheduling, the timer, dispatching, creation and termination of sessions, main memory management and command interpretation. There are monitor services which allow user and less privileged portions of the system access to I/O and timer functions, etc. (See 3.3 for a description of these services). In addition, there are monitor services which allow privileged commands access to system data.

All unused terminals are monitored by a special nucleus process, which upon reading a line from a terminal verifies that a valid user has issued a logon command. In this case, a mover is created, and that mover is initialized to execute the command interpreter.

The scheduler selects a subset of non-waiting processes as eligible to run. Multiprogramming takes place only over this subset, under control of the dispatcher, until the scheduler alters the subset of eligible processes. This subset is altered when a process terminates or is created, when demand on the memory hierarchy changes, or when a member of the current set of eligible processes exceeds a bound on the amount of service received. When a new process becomes eligible, memory management may be invoked to make room for the process, either by moving main storage areas for running processes toward the lowest numbered memory locations in order to create a large block of unused main memory, or by copying the memory of ineligible processes onto secondary storage.

The dispatcher allocates CPUs to eligible processes. A clock and an interval timer are used to control the amount of uninterrupted CPU time given to a process; the dispatcher generally gives short bursts or quanta of CPU service to the eligible programs in round-robin fashion.

The command interpreter for our time sharing system (which comes into play once logon has been successfully completed) uses the user's vector of command libraries to determine which libraries to search for the file defining a particular command. If a command file consists of further commands rather than a program, then the command interpreter is invoked recursively. If a command file is a program, a block of contiguous memory of appropriate size is allocated to the user, the system loader moves the program into this block, and control is given to the program, which is initially given high priority for scheduling consideration. The base and bound for the program are set so that direct reference can only be made to main memory allocated to the program. If the program was loaded in response to a privileged command, then the program has additional monitor services available to manipulate system data.

5.5.2 Privileged System Commands

Privileged system commands include 'logoff', the file system commands described in 5.3, and memory hierarchy management commands. These commands, being privileged, can use PSETL disable blocks to guarantee orderly use of shared sets, such as the catalogue. In this implementation, these commands are interpreted by subprograms to the command analyzer which are in the scope of the system nucleus. Thus, the system data structures (see 2.4.1, 3.5.1, and 5.7.1) are accessible to these subprograms.

(In a more general implementation, the privileged commands could be executed by loading and executing system modules much in the same manner as user modules are loaded and executed. In such a case, the addressing structure of our machine would make the system data unaddressable by these modules, and special monitor commands, usable only by privileged commands, would be required to enable these system modules to access and modify system structures.)

5.5.3 Non-privileged System Commands

The non-privileged system commands provide utility services to users but do not differ from ordinary user-written programs. Examples are compilers, sorting routines, and file editors. Because such utilities do not have any characteristics peculiar to operating systems, these programs will not be further discussed or shown. It should be noted, however, that these utilities are often a user's principal interface with the interactive system. The success of a time sharing system is strongly affected by the quality of utilities of this type. File editors, in particular, are very important utility programs.

5.6 Remarks on the Interactive System

In some respects, the code for the interactive system is easier to follow than that for the two previous systems. For the non-interactive systems, JCL is used to specify a **sequence** of job steps and the interrelationships among the job steps are established through the file system. The interactive system is oriented more toward handling single commands, and the mechanisms which relate file references from one command to the next are not as elaborate. Scheduling decisions are also simpler, since optimization of resource usage over substantial periods of time is not being attempted. Preemptive assignment of main memory removes the necessity of long range scheduling for that resource. Dispatching of processes does become more sensitive, since processes are originally assumed to be short running and given priority on that basis. If the assumptions of small resource requirements proves incorrect, dispatching and scheduling must be adjusted.

5.6.1 Logon

The logon process operates from a workqueue on which idle terminals are queued. The process initializes itself by putting all the terminals on its workqueue.

Reads are issued at label 'idle' for all idle terminals. When the process is notified of completion of the read by means of an I/O interrupt at 'termfixup', it checks for a valid user logon. Not only must the user's identification be valid, but he must not already be logged on. If a valid user logon command has been accepted, he must next enter his password. The terminal handling to this point has employed asynchronous basic monitor functions (see 3.3) so that one process can control multiple devices without any implicit waiting for completion of terminal read operations. The bulk of terminal input processing then takes place in the I/O fixup routine. After a user has identified himself and given the correct password, a process which accepts and interprets commands for that user's session is created. The log-on process detaches the user's terminal from itself.

5.6.2 Command Analyzer

For each process (user session) executing the command analyzer, an initial block searches the system's profile library to get the defaults and library searchorder for the user on whose behalf the

process is running. Initialization ends by executing a 'setup' command;this automatically executes the user's setup procedure if one exists, after logon.

In the command analyzer routine 'source' is a vector of input file identifiers, the first component of which is the current input file. This vector serves as a pushdown stack of file names, new names being added to the head of the stack whenever a command invokes a file of commands; names are dropped from the head of the vector whenever the end of a file of commands is reached.

With the exception of the simulated read of the setup command, all command reading takes place at the label 'readdata'. If the command is read from a command file rather than the terminal, references to parameters are replaced by their values. Parameters are encoded by a '$' followed by an integer. The integer describes which of the arguments of the command which invoked the current input source is to be substituted. The string of arguments from that command are saved as the first component of 'argumentstack'.

Analysis of a command begins at the label 'lookup'. If the command name is that of a built-in command, then the code corresponding to that command is executed. Otherwise, the libraries pertaining to commands, modules, and text are searched, in that order, for an object whose name agrees with the command name.

If a command-file is found whose name agrees with the command name, then at label 'foundc' that file becomes the new input source for the command analyzer, and the remainder of the original command gets saved as the first component of 'argumentstack' to substitute for parameters encountered in commands read from the new source.

If a command name is found to refer to an executable main memory image (we shall call these images 'module files'), a block of main memory of sufficient size is reserved for the module, and the module is read into main memory. A state description for a user program is created, and relocation boundaries are set up to restrict memory references to the block of storage just obtained for the module. A process for the module is initiated, and the command analyzer (that is, the command analyzer instance working for the user whose command was just read) waits for an item to be placed on its workqueue.

There are two ways in which an item can become enqueued on the command analyzer's workqueue. When a user process terminates execution (see the 'abend' and 'endstep' monitor calls at the end of the code of Sec. 5.7.2), the terminating condition is enqueued on the command analyzer and the ancestor of the terminating condition is set to **nl**.

When the command analyzer is being invoked recursively by a user process, then the arguments for the command are enqueued on the command analyzer on behalf of the process which issued the monitor call (see section 5.2). The current user process is suspended, and a new user process is set up. Then (at 'readdata'), after the recursive command has been obeyed, the former user process is reactivated.

To prevent endless recursion of commands and to limit the amount of secondary storage space which a user can occupy, a maximum of ten levels of recursion are allowed. If an eleventh is requested, execution of all levels of recursion terminate, after which the user may enter a new command from his terminal.

If a text-file's name agrees with the command name, then an object file (that is, the machine language output of a language processor) exists to carry out the command. The distinction between a text file and an module file is this: the module is an image of the initial state of main memory of code which can be executed. A text-file is the output of a compiler, and may have to be combined with compiler output for missing procedures before an executable memory image exists. This processing is carried out by the **loader**. When a command refers to a text file, the code at label 'foundt' prepares to load the system module corresponding to the loader, and then causes the loader module to be fetched. The loader, when it executes (not shown here) takes the command data as input to determine which text files are to be loaded.

5.6.3 Main Storage Management

The procedure **getblock** is a queued subroutine which finds and allocates a block of main memory to a process. If the size of the requested block is too large, then **getblock** immediately returns Ω.

The **getblock** procedure uses three vectors to describe storage allocation. The vector memloc gives the starting locations for all memory blocks, the vector memsize gives the sizes of the blocks, and the vector memown identifies the processes which own the blocks.

First **getblock** attempts to satisfy the request without relocating any blocks of memory. If an unowned block of at least the requested size exists, a memory block of the required size is allocated to the process specified by **P**.

If a sufficiently large block of contiguous storage does not already exist, then the disabled block following the label 'addup' computes the total amount of unused storage, and if enough storage exists, then the allocated blocks of memory are moved to lower addresses in order to collect all the unassigned memory into a contiguous block. As each assigned memory block is moved, the relocation portion of the environment of the process associated with that storage block is adjusted to reflect the new base of the process's main storage.

If insufficient free storage exists even after compacting used storage, then those blocks of main storage belonging to processes which are considered ineligible for scheduling are marked as belonging to the requesting process, and the contents of those blocks are saved in secondary storage. The statement:

must=∃y(j)∈extstore | y **eq** pr;

sets j to the number of the block in secondary storage which is devoted to holding the memory image of a process pr which is being displaced. The string 'secondarystorage' represents the system file which holds the memory images of processes which have been preempted from main memory, and **extstore** is the map which determines where in the secondary storage area a process's memory image is saved.

In any case, once a block of storage has been found, control reaches the block of code labelled 'final', where the memory maps are updated to reflect the assignment just made.

5.6.4 Scheduling and Timing

An eligible process is one which can run given main memory space and control of the CPU, but is not momentarily being considered for control of the CPU by the dispatcher. The scheduler uses two work queues of eligible processes. The first queue, 'hiprio', contains the identifiers of processes which the system assumes are of high priority. System processes, and processes which had been started as the result of commands issued from a terminal but which have not used more than a predetermined amount of resources are queued in 'hiprio'. Processes which have exceeded the threshhold which separates quick running from long running interactions are queued in 'lowprio'.

The scheduler waits until 'hiprio' and 'lowprio' are not simultaneously empty. It then calls a subroutine, **sched**, passing the highest priority, non-empty work queue as an argument. Note that **sched** and **getblock** must not interfere with each other during execution. Without care, this could occur, since **getblock** looks for eligible processes which have main memory blocks assigned, and seizes these blocks to carry out its function, while **sched** must determine if the process it has selected is already in main memory. To avoid interference, both subroutines are queued on the same facility, 'blockflag'. If **sched** determines that the selected process is not in main memory, it calls **getblock** to reserve a memory area of sufficient size, and adjusts the relocation portion of the selected process's environment for its new memory assignment. Finally, **sched** resets the time used in the current time slice to zero, and it places the scheduled process at the end of the CPU's workqueue. The scheduler attempts to schedule more processes so long as processes remain in the queues hiprio or lowprio, and so long as memory is available.

Control goes to the block of code labelled 'timexpt' whenever there is a timer interrupt. If a user program was interrupted, the time used since the interrupted process has been dispatched is computed, and added to the total time used during the current timing period. If this total exceeds the time "quantum" allowed for the process then the CPU time used during the current timing period is added to the total CPU time used by the process. If additional quanta remain in the time slice, a new timing period is begun for the process (usednow=0;), and the process is placed at the end of the dispatcher's queue. If no quanta remain, then the process's time slice has ended. Subsequent timeslices will consist of one second, and the process is placed at the end of the low priority queue of eligible processes.

The timer interrupt handler then terminates, which forces control to the dispatcher, which will, in a round robin fashion, select the next dispatchable process. The purpose of dividing time slices into quanta is to guarantee that among dispatchable processes, none must wait an intolerable amount of time before gaining access to the CPU. Thus a trivial request which can execute in one quantum will always appear to respond quickly (once it is made dispatchable).

5.7 Coded Interactive System

5.7.1 System Structures

In the code which follows, we assume the existence of the system data structures which were previously described in Section 3.5.1. Descriptions of additional system structures used by the interactive system follow below.

usernamegiven: In 'log-on', usernamegiven is a map from terminals on which a valid log-on has been received, to user names. The map is defined only for those terminals still waiting for a password.

waitingset: In 'log-on', **waitingset** is the set of terminals for which a 'read' command had been issued, but on which the read has not yet been completed.

bufferfor: In 'log-on', **bufferfor** is a map from terminals to corresponding buffers which the terminals use for I/O in conjunction with monitor service commands.

extstore: **extstore** is a vector whose components are user processes. The position of a process in **extstore** determines where in secondary storage the process's memory image is kept, when it is necessary to swap the process's main memory image to secondary storage.

msg: For $u \in$ signedon, msg(u) is a vector of character strings. Each character string is a message to be typed at the user's terminal, when the current command completes execution.

memown: At any given time, main memory is partitioned into a number of blocks of contiguous storage. **memown** is a vector having as many components as there are blocks. The k^{th} component of **memown** gives the process to which the k^{th} blcok is assigned. If the block is unassigned, the component is Ω.

memsize: **memsize** is a vector which gives the sizes of the blocks of main memory.

memloc: **memloc** is a vector which gives the starting locations of the blocks of main memory.

size: For each user process p, size(p) is the amount of main memory which that process requires.

quantum: For each user u, quantum(u) gives the amount of uninterrupted CPU time which u's processes may use before the dispatcher will give the CPU to another process which is on the dispatcher's work queue.

mquant: For each user u, mquant(u) gives the number of quanta which u will be allowed before being removed from the dispatcher's work queue.

nquant: For each user u, nquant(u) gives the number of quanta which u has used (including the present, unfinished quantum), since being put on the dispatcher's work queue.

usednow: For each user u, usednow(u) indicates how much time of the current quantum has been used by u's processes.

5.7.2 The Code

```
/*
```

Catalogue of Routines


```
*/
```

```
scope interactivesystem;
macro establish(ufn,pfn,filevar);
    /*For user file ufn, a program file name pfn is established for the running process, and filevar
    becomes a file variable representing the file*/
    assign(<CPUcontrol,newat> is pfn>,ufn,standardfixup);
    filevar=<open pfn,1>;
endm establish;

macro processparameter;
/*reference a processes initial argument*/
    info(initialvar(state(CPUcontrol)));
endm processparameter;

macro filesize;
/*default size of standard system-created files*/
    100
endm filesize;

macro primarysource;
/*for the input reader, reference to main input device*/
    hd(source(#source))
endm primarysource;

macro primaryposition;
/*for input spool position in the primary file next to be read*/
    (source(#source))(2)
endm primaryposition;

macro currentsource;
/*for input spool, the current input program-file name*/
    hd hd source
endm currentsource;

macro currentposition;
/*for input spool, the current position in the input file*/
    (hd source)(2)
endm currentposition;

macro readingprimarysource;
/*true iff current source is the system input device*/
    #source eq 1
endm readingprimarysource;

macro thisprocess;
        /*A more mnemonic way for a process to refer to its own process identifier.*/
    CPUcontrol
endm thisprocess;

macro deviceaddress(pfn); /*device on which program file resides*/
    catalogue(userfile(pfn))('deviceaddress')
endm deviceaddress;

macro dataread; /*to access data read from terminal*/
    buffer bufferfor(terminalname)
endm dataread;

macro devicepart(x); /*program file causing interrupt.*/
    x(1) /*first component of x, cf. 3.3.2*/
endm devicepart;
```

```
macro writetoterminal(terminal, message);
    buffer bufferfor(terminal) = message; /*Set up terminal buffer*/
    monitor ('write', terminal, bufferfor(terminal));
endm writetoterminal;

macro maxmem;
    10000 /*Machine and operating system dependent -maximum size of a main storage block
        which can be assigned to a user process.*/
endm maxmem;

macro freesecondarystorage(p)  /*Releases secondary storage held by process p.*/
    (disable)
        must= 1 ≤ ∃j ≤ #extstore | extstore(j) eq p;
            /*The above statement sets j to the position in extstore corresponding to the user
            process p.  If p is a user process, then j will be defined.*/
        extstore(j) = Ω;  /*Release secondary storage space.*/
    end disable;
endm freesecondarystorage;

macro tick; /*short time slice quantum*/
    100000 /*0.1 sec for CPU with a 1μs. timer.*/
endm tick;

macro onesecond; /*The number of timer units in one second*/
    1000000  /*assuming a CPU with a 1μs. timer.*/
endm timer;

macro xquant;  /*Number of quanta we will allow a process at high priority*/
    5
endm xquant;

macro moverpart(pr) /*extract mover i.d. from a process i.d.*/
    hd pr
endm moverpart;

macro suspendedprocess;
        /*Most recently suspended user process, gotten from top of procvec.*/
    procvec(1)(1)
endm suspendedprocess;

macro recursiondepth;
        /*Depth of recursion when most recently suspended user process started.*/
    procvec(1)(2)
endm recursiondepth;

macro pair(x);
        /*This macro generates a boolean expression which is true if and only if x is a tuple of two
        elements.*/
    (type x eq tuple) and #x eq 2
endm pair;

/*The global variables declard below are described in section 3.5.1*/
global iowait, mainstate, busystatus, savecause, filewait, fixup logicalposition, channels, units,
    volumes, programfiles, devices, type, userprogs, steptimeleft, ready, mounted, deviceaddress,
    symbolicfile, operational, users, owners, defaults, catalogue, maxprio, esttime, timeleft,
    iofixup, workset, userfile, newallocation, budget;

/*The global variables declared below are described in section 5.7.1.*/
global extstore, msgon, msg, secondarystorage;
```

```
/*  ************************************  */
/*                LOG-ON                  */
/*  ************************************  */

scope logon;

    /*Initialization of the logon procedure. First, all terminals in the system are put onto
    logon's workqueue. In the main body of 'logon' for every item on it's workqueue, the
    'logon' procedure starts a logon sequence by issuing a read command to the terminal. After
    initialization, terminals are returned to logon's workqueue whenever a user ends a terminal
    session.*/
  (∀t∈devices | type(t) eq 'terminal')
        /*Enqueue terminal on this process*/
    putlast(thisprocess,t);
  end ∀t;
    /*Initialize map from terminals to user-names. This map only has entries for users who
    have given a logon command, but have not yet given their password.*/
  usernamegiven=nl;
  waitingset=nl; /*Initialize the set of terminals for which a log-on related read operation is in
    progress.*/

    /*The command interpreter has the same environment as log-on, but starts at label
    'command'. We set up a state object for the command interpreter, which we split to for
    each user who successfully logs on. See the code following the label 'checkpassword'.*/
  commandenv=<thisprocess,state(thisprocess)>;
  loctr(commandenv)=command;

    /* Main loop of Log-on */

        /*The log-on routine is inactive so long as no terminals are on it's workqueue, or all its
        read operations are incomplete. It is necessary to wait for the alternation of conditions,
        for if we only waited for the first of them, i.e., for items to be stacked on the
        workqueue, then the log-on process could remain in waitset (cf. 2.4.3.1), even after a
        read operation. On the other hand, if the await is satisfied by a read operation being
        completed, then the terminal fixup routine at label 'termfixup' would be executed as
        soon as the log-on process is removed from waitset and is made dispatchable, by virtue
        of the call to 'assign' below. The statement following the await would be executed only
        after the fixup routine terminates.*/
idle: await (#workset{thisprocess} gt 0) or (waitingset-busy ne nl);
        /*For each idle terminal, we issue a read via monitor call, so that the read operations
        are overlapped. The 'assign', executed below, specifies that at the conclusion of a
        'read', control will be forced to 'termfixup'.*/
  if #workset{thisprocess} eq 0 then
        /*No new terminals were enqueued on log-on. Interrupts from read operations caused
        the set (waitingset – busy) to be non-null. These interrupts were processed by a forced
        transfer to 'termfixup' as soon as the waiting state ended. Thus, workset{thisprocess}
        can be empty even when the await at 'idle' has been satisfied.*/
    go to idle;
  end if;
  terminaltologon=getfirst(thisprocess); /*idle terminal id*/
    /*Set up program file name for terminal which desires logon. All such terminals will use
    the monitor I/O services (cf. 3.3.2), so that the logon process does not have to logically
    wait for a read to be completed. Whenever a read operation does terminate, control goes
    to 'termfixup', where the data read gets processed.*/
  assign(<thisprocess, newat> is tfn,terminaltologon,termfixup);
  bufferfor(tfn)=newat; /*Identifies buffer which 'tfn' will use for I/O.*/
  errorcount(tfn)=0; /*initialize consecutive error count to 0*/
  tfn in waitingset; /*Indicate that x is waiting to read input for log-on.*/
  monitor('read',tfn,bufferfor(tfn)); /*read probable logon command*/
  go to idle;
```

```
    /* terminal interrupt handler.
        Processing to check logon command and password takes place in interrupt routine.
    Note that since interrupt routines are logically non-interruptable, there can be no race
    conditions in the following code, even if several users attempt to log-on with the same user
    i.d.*/

termfixup:
    /*We define a function usernamegiven from terminals to user names. It is defined only for
    terminals which have received a valid logon-command, and are expecting a password.*/
  monitor('iointerrupt',interruptbuf); /*get cause of interrupt*/
  interruptreport = buffer interruptbuf; /*See 3.3.2 for a discussion of the buffer function.*/
  terminalname=devicepart(interruptreport); /*programfilename which caused interrupt*/
    /*cf. page 113 for the macro 'devicepart'*/
  terminalname out waitingset; /*remove terminal from the set of terminals
                which have a log-on read outstanding.*/
    /*'dataread' macro extracts line just read from terminal's buffer*/
  image=dataread;
    /*'error' is a machine dependent macro which generates a true boolean expression if and
    only if the interrupt report indicates an error.*/
  if error(interruptreport) then
    errorcount(terminalname)=errorcount(terminalname)+1;
    if errorcount(terminalname) le 10 then
      writetoterminal(terminalname,'transmission error, repeat line');
    else /*Excessively many errors.*/
        /*Try to warn user that his terminal is not working properly.*/
      writetoterminal(terminalname,'bad terminal or line, removed for CE');
      operational(deviceaddress(terminalname))=false; /*cf. 3.5.1.4 for 'operational'*/
        /*Warn operator that terminal requires attention.*/
      operatormessage('terminal'+deviceaddress(terminalname)+'requires attention');
      unhook(terminalname); /*Release file name representing terminal*/
      usernamegiven(terminalname)=Ω; /*No longer expecting a password.*/
      monitor('endfixup');
    end if;
  else if usernamegiven(terminalname) ne Ω then
    go to checkpassword; /*password is expected*/
  else if not (pair(image) and hd image eq 'logon') then
    writetoterminal(terminalname,'improper logon');
  else /*Attempt first step of logon operation -- verify that user i.d. is valid.*/
    username=image(2); /*identify user*/
    if username ∈ (users – signedon) then
        /*We have a valid user, who is not already logged on. Note that since I/O fixups
        are not logically interruptable, even if several terminals simultaneously issue logons
        with the same user i.d., only the first terminal to have its interrupt processed will be
        considered valid!*/
      username in signedon;
      usernamegiven(terminalname)=username; /*indicate next read is to get password*/
    else
      writetoterminal(terminalname,if username ∈ users then 'already signed on'
            else 'invalid user-id');
    end if username;
  end if error;
reread:
  monitor('read',terminalname,bufferfor(terminalname));
  terminalname in waitingset; /*Indicate that a 'read' is again outstanding.*/
  monitor('endfixup');

checkpassword:
  errorcount(terminalname)=0; /*Reset error count, since previous read was okay.*/
    /*Check the password.*/
  if image ne password(usernamegiven(terminalname) is username) then
```

```
            /*Wrong password -- start the logon sequence all over.*/
        writetoterminal(terminalname,'bad password, re-issue logon and try again.');
            /*Having had a bad logon attempt, we set up the terminal to start a new logon se-
            quence.*/
        putlast(thisprocess,deviceaddress(terminalname));
            /*usernamegiven(terminalname) will be dropped below.*/
        username out signedon;  /*Another logon for 'username' can now be accepted*/
    else
            /*Password is okay.*/
        moverid=newat; /*mover-id for user*/
        owner(moverid)=username;
        processpart(commandenv)=<moverid,newat>;
            /*Here we are assuming that 'commandenv' is the description of the command
            analyzer's initial environment.  Only the processpart must be given a unique initial
            value to start the command analyzer for the new user who has just logged-on.*/
        split to commandenv(deviceaddress(terminalname)) for thisprocess;
    end if;
    usernamegiven(terminalname)=Ω; /*Remove from map -- not waiting for password.*/
    unhook(terminalname);
    monitor('endfixup');
end logon;

                /*  ***************************  */
                /*          COMMAND ANALYZER     */
                /*  ***************************  */

scope commandanalyzer;

    /*For each user who has successfully logged on, a command analyzer process has been
    created.  This process reads commands from the user's terminal or from a command-macro
    library, and interprets these commands by:
        1.  obeying the command directly,
        2.  loading a program,
        3.  accepting commands from a macro-library.
    When executing programs (case 2), a user process, of lower privilege, is created.  This
    less-privileged process controls the CPU during the execution of the user's programs.  A
    user program can also execute commands by using the monitor services (cf. 5.2.)*/

                /*  ***************************  */
                /*        Initialize User Defaults */
                /*  ***************************  */

commandsetup:
    terminalid=processparameter;  /*get user's terminal identifier*/
        /*Establish 'terminal' as a file name for the user's terminal.*/
    assign(<thisprocess,newat> is terminal,terminalid,standardfixup);
    movid=moverpart(thisprocess);
    userprocess=<movid,newat>; /*Identifier for lower privileged process which will run user
        programs.  All processes for this user session will belong to the mover movid.*/
    userid=owner(movid);  /*Derive the user's identification.*/
        /*Each user is assumed to have a file called 'profile', which contains two structures:  the
        user's defaults for required parameters, and the user's library search order.*/
    if <userid,'profile'> ∈ datafiles then
            /*The user has defined  a profile file.*/
        establish(<userid,'profile'>,profileopenclose,userprofile);
    else
            /*The user doesn't have a profile file.  Use the system's.*/
        establish(<'system','profile'>, profileopenclose, userprofile);
    end if;
```

```
userprofile read default;
userprofile read searchorder;
unhook(profileopenclose); /*user profile is no longer needed*/
    /*Initialize the stack of input sources used for recursive invocations of the command
    analyzer. 'source' is a vector whose components are pairs, each pair giving the program file
    name of a file from which commands are being read and the location within the file from
    which the next command is to be read. Implementation note: In this code, items are added
    to and taken from the front of vectors which represent stacks.*/
source=<<terminal,1>>;
    /*'argumentstack' is a vector whose components are vectors of arguments to be used in
    macro-command expansion. When a macro is being expanded, the components of
    argumentstack(1) are used for parameter substitution. See the code below, near the label
    'moreparms'.*/
argumentstack=<nult>;
    /*Initialize procvec, which is a vector of pairs of the form <p,n>. Each p represents a
    process, initiated by the command analyzer, which has been suspended while a command
    issued by it (i.e. p) is being executed. n represents the number of recursive invocations of
    the command analyzer which had already been issued when process p issued the command
    being interpreted. n is used near 'readdata', below, to determine when p's command has
    been completed.*/
procvec=nult;
infile=<open currentsource,1>; /*currentsource is a macro*/
    /*extstore is an auxiliary data structure used to map active user processes into secondary
    storage. When an active process must be swapped out into secondary memory, the user's
    position in extstore determines the location in secondary storage reserved for the process
    image. The form of extstore is a vector of process identifiers. The position of a process-id
    in the vector is used by the 'getblock' routine to determine where in secondary storage the
    user's swap area is.*/
(disable)
    if extstore eq Ω then
        extstore=<userprocess>;
    else if (1<∃j<#extstore+1 | extstore(j)eq Ω) then
            /*Note that the preceeding 'if' cannot fail to be satisfied.*/
        extstore(j)=userprocess;
    end if;
end disable;
    /*In the main loop of the command analyzer, 'image' will hold the line read from the
    terminal. We initialize it to the string 'setup' so that the user's setup command is executed
    immediately. cf. Sec. 5.2.1.*/
image=<'setup'>; /*Simulate reading of 'setup' command*/
go to lookup; /*to begin handling the 'setup' command forced into the input stream just
    above.*/

            /*  ***************************  */
            /*       Main Loop of Command Analyzer       */
            /*  ***************************  */

readdata:
    /* At this point, a prior command has just completed, and we read the next command
    from the topmost command file or from the terminal, substituting parameters to get its
    'true' form if necessary.

      Before reading, however, we must determine whether a recursive use of the command
    analyzer, invoked by a running program, has been completed.*/
if procvec ne nult then /*recursive use of command analyzer*/
        /*Recall that the top element of procvec gives n, the number of recursive invocations
        of the command analyzer at the time when the most recent program-initiated invoca-
        tion of the command analyzer was received. If the level of recursion is now n, then the
        interpretation of a run-time command is completed. The current temporary user
```

```
            process may be released, and the suspended process, also given by the top element of
            procvec, resumes operation.*/
        if #source eq recursiondepth then /*Command completed.  (See macro defs.)*/
            kill userprocess;
                /*Release swap-space for killed process.*/
            freesecondarystorage(j);  /*See macro definitions for next two statements.*/
                /*Restart suspended process*/
            userprocess=suspendedprocess;
            procvec=procvec(2:); /*Remove top element from stack.*/
            putlast(CPU,userprocess); /*Resume the suspended process.*/
            go to waiting;
        end if;
    end if;

        /*Read the next command, but first test for messages stacked waiting to be sent to the
        terminal user.  If there are such messages, they will be transmitted to the user just before
        we accept additional from the user's terminal.*/
    if msgon and readingprimarysource then  /*if user is willing to receive messages*/
        (while #msg(userid) gt 0)  /*and there are any messages,*/
            writetoterminal(terminal,hd msg(userid));  /*transmit them!*/
            msg(userid) = tl msg(userid);
        end while;
    end if;

                /*  ***************************  */
                /*            Read Next Command  */
                /*  ***************************  */

    infile read image;
    if image eq Ω then  /*test for end-of-file*/
        if readingprimarysource then  /*EOF from terminal*/
            image=<'logoff'>;  /*force end of session*/
            go to lookup;
        else
                /*End-of-file expanding a command macro, go back to reading previous source file,
                or from the terminal.*/
            unhook(currentsource);
            source=tl source;
                /*Drop the file on which the EOF was encountered, and re-open the file which
                contained the macro which was just expanded.*/
            infile=<open currentsource,currentposition>;
            argumentstack=argumentstack(2:);  /*remove stacked parameters*/
            go to readdata;  /*resume reading from previous file*/
        end if;
    end if;
    if not readingprimarysource then
            /*Execute subroutine to substitue values for parameters in command just read.*/
        substparams;
    end if;

                /*  ***************************  */
                /*      Command Examination & Execution  */
                /*  ***************************  */

lookup:
    if type image ne tuple then
            /*Command has wrong format*/
        go to commanderror;
    end if type;
    command = hd image;
```

```
        /*First, we look for built-in commands. All built-in commands are treated as subprograms
        to the command analyzer, and they return character strings, which are transmitted to the
        user as a response to his command.*/
short:
    if {<'share',sharecom>,
        <'file',filecom>,
        <'define',definecom>,
        <'permit',permitcom>,
        <'searchorder',searching>,
        <'logoff',logoffcom>,
        <'message',messagecom>,
        <'holdmsg',holdmsg>,
        <'releasemsg',releasemsg>} (command) is routine ne Ω then
            /*Command found. Execute built-in command and write the resulting message to
            the user's terminal if the command was issued directly from the user's terminal. All
            built in commands are functions callable by the command analyzer. These func-
            tions return character strings.*/
        result=routine(userid,userprocess, tl image); /*Interpret command*/
        if readingprimarysource then
            /*If the command was issued from the terminal, then write the resulting
            message to the user's terminal. If there were no errors, the resulting
            message is nulc.*/
            terminal write result;
        end if;
        go to readdata; /*Get next command.*/
    end if;

        /*Not a built-in command. The command libraries are searched next (for possible macro
        expansion), then the module libraries are searched (for a file containing an executable
        program), and finally the text library is searched (for compiler output, which must first be
        processed by a loader (not shown here) to supply missing subroutines).*/
    (∀libtype(i) ∈ <'command','module','text'>, library(j) ∈ searchorder(libtype))
        if library eq '*' then
            /*Use the system library. A share command is needed to make the system library
            accessible to this process, which is running for a user's mover. cf. Sec 5.4.3.*/
            monitor('share','library.'+libtype,system,'library.'+libtype);
            establish('library.'+libtype,tempopenclose,templib);
        else
            establish('library.'+library,tempopenclose,templib);
        end if;
        templib read list;
        if type list ne set then
            /*Ignore the library if it is in the wrong format.*/
            continue ∀ libtype;
        end if type;
        unhook(tempopenclose);
            /*See if the required command is in the list of library routines.*/
        if(∃commandfilename ∈ list | commandfilename eq (libtype+'.'+command)) then
            /*library file found*/
            go to {<'command',foundc>,
                <'module',foundm>,
                <'text',foundt>} (libtype);
        end if;
    end ∀libtype;

        /*The command cannot be found. It is neither built in, nor a macro command, nor a
        module, nor a text file. Warn the user, and go on to the next command.*/
commanderror:
    terminal write 'illegal command:' + image + 'from file', currentsource,'. ignored';
    go to readdata;
```

```
foundc: /*Enter here when a command refers to a file of subcommands.*/
        /* read commands from the command file named by the just-received command until an
        end-of-file is detected at readdata.*/
    currentposition=infile(2); /*remember position within current file*/
        /*Redefine 'infile' to refer to the library file which holds the macro-expansion of the
        command just read.*/
    establish(commandfilename,commandopenclose,infile);
    source=<<commandopenclose,1>>+source; /*Update the stack of input files.*/
        /*Put the parameters from the current command on top of the parameter stack.*/
    argumentstack=<image(2:)>+argumentstack;
    go to readdata;

foundm: /*Enter here when a command refers to an executable module.*/
    assign (<thisprocess,newat> is tempopenclose,commandfilename,standardfixup);
fetchmodule: modulefile=<open tempopenclose,1>;
        /*We assume that modules are self describing files of two records. The second record
        contains the code to be loaded and executed; the first record gives the length of the
        second.*/
    modulefile read bound;
        /*'maxmem' is a macro giving the size of the largest block of contiguous storage available
        for assignment. It is machine dependent.*/
    if bound eq Ω or bound lt 1 or bound gt maxmem then
        terminal write 'bad module size'+command+dec bound;
        go to readdata;
    end if;
        /*Secure a memory block for the module. This is always possible in principle, since we
        have already checked that the amount of space being requested is not excessive, and no
        other program will be allocated space until this request is fulfilled. We could test 'base ne
        Ω' after the next statement as a software check.*/
    base=getblock(bound,userprocess);
(disable)
        /*Construct initial state for the program to be run.*/
    state(userprocess)=userstate; /*Standard non-privileged state.*/
        /*Location and size of allocated main memory into environment.*/
    relocate(state(userprocess))=(base,bound);
        /*Read module into assigned memory locations. Recall that memory is a vector represent-
        ing all of main memory. cf. section 5.3*/
    modulefile read memory(base:bound);
        /*Initialize timing information. xquant is a macro giving the number of times the user-
        process is to be scheduled at high priority in a round robin fashion at the start of a com-
        mand. Each time scheduled, the process will be allowed one 'tick', where 'tick' is defined
        via a macro. A typical value might be 0.1 sec.*/
    if readingprimarysource then
            /*The command being interpreted was entered directly from the user's terminal. Start
            the user process at high priority. For other commands, invoked recursively or by
            running processes, this then-clause is skipped, and the previous timing parameters
            remain in effect. Thus, if the most recently issued user command has used a great deal
            of CPU time (0.5 seconds, in this version), the user's processes will continue to run at
            low priority, until he enters his next command from the terminal.*/
        mquant(movid)=xquant; /*See macros for 'xquant'*/
        quantum(movid)=tick;
        nquant(movid)=1; /*Indicate this is the first quantum*/
        usednow(movid)=0; /*and that no time has been used so far*/
        putlast(hiprio,userprocess); /*put process at end of scheduler's hi priority list.*/
    end if readingprimarysource;
        /*We assume that execution starts at relative location 0.*/
    loctr(state(userprocess))=0;
    size(userprocess)=bound;
            /*Place ancestor and initial parameters into the user process environment. Include the
            terminal i.d. in the set of initial parameters.*/
```

```
    split to userprocess(<terminal>+image(2:)) for thisprocess;
end disable;
    unhook(tempopenclose); /*Release file containing program module.*/
        /*Wait for the user-process to terminate. We could, at this point, associate, with the user's
        terminal, a fixup routine which, on recognizing an attention request from the terminal,
        reads a line from the terminal to determine whether the execution of the user-command
        should be preemptorily terminated.*/
waiting:
    await getfirst(thisprocess) is fault ne Ω;
    if ancestor(fault) eq nl then
            /*Command is complete. Read next command.*/
        go to readdata;
    else
            /*Interpret command for running program issued via a monitor call.*/
        if #procvec ge 10 then
                /*Recursion too deep -- notify user.*/
            terminal write 'recursion too deep, command terminated.';
                /*Release all user processes resulting from recursive use of the command analyz-
                er.*/
            (while #procvec gt 1) /*Release all user processes*/
                kill userprocess;
                freesecondarystorage(userprocess); /*Give up swap space for process.*/
                userprocess=suspendedprocess;
                procvec=procvec(2:);
            end while;
            if #source gt 1 then
                unhook(currentsource);
                    /*When recursion is too deep, the terminal becomes the input source again, and
                    all other input sources, which are in the source stack, are dropped. Remember
                    that the terminal is on the bottom of the stack, i.e., the last item in our indexing
                    scheme for stacks.*/
                source=source(#source:1);
                infile=<open curentsource, currentposition>;
                argumentstack=<nult>;
            end if;
            go to readdata;
        end if;

            /*Recursion is not too deep.*/
        image=info(fault); /*The actual command.*/
        procvec=<userprocess,#source>+procvec;
        userprocess=<movid,newat>; /*New process to execute command, if needed.*/
        (disable) /*Get additional storage for new process.*/
            if (1≤ ∃j ≤ #extstore+1 | extstore(j) eq Ω) then
                    /*Note: The preceeding 'if' cannot fail to be satisfied.*/
                extstore(j)=userprocess;
            end if;
        end disable;
        substparams; /*replace parameters with values/
        go to lookup; /*Examine and interpret command*/
    end if;

foundt:
        /*If the command name refers to a compiler output file, the loader is run as the user's
        program. The loader, whose internal details are not shown here, will load the text corre-
        sponding to the command, supply missing routines (see a sketch of this in the uniprogram-
        med system's job control language interpreter) and then give control to the module just
        constructed.*/
    monitor('share','module.loader',system,'module.loader');
    assign(<thisprocess,newat>,<userid,'module.loader'>,standardfixup);
```

```
    image=<loader>+image; /*so that code at foundmodule
                sets up data for the loader correctly.*/
    go to fetchmodule;

define substparams;
            /*This subprogram substitues values taken from the top of the argument stack for
            parameters appearing in 'image'.*/
moreparms:
    if(∃y(i) ∈ image, n(j) ∈ y | n eq '$') then
            /*  possible parameter, field starts with '$' */
        k=j+1;  /*k will point beyond parameter*/
        m=0; /*Develop m into parameter number.*/
        (while y(k) ∈ {0,1,2,3,4,5,6,7,8,9} doing k=k+1;)
            m=10*m+y(k); /* develop parameter number*/
        end while;
        if m gt 0 and m le #argumentstack(1) then
                /*valid reference to a parameter, substitute its value*/
            image(i)=y(1:j-1)+argumentstack(1)(m)+y(k:);
        else/*nonvalid parameter, disguise it 'til later.*/
            image(i)(j)=er;
        end if;
        go to moreparms;  /*Look for more parameters*/
    end if;
        /* Now remove any ers that were used to temporarily replace $ */
    (∀y(i) ∈ image, n(j) ∈ y | n eq er)
        image(i)(j)='$';
    end ∀;
    return;
    end substparams;


                /*  ***************************  */
                /*        Linkage to Command Interpreters        */
                /*  ***************************  */

    /* The following routines interpret file handling commands. Their principlal function is to
    prefix userfile names with user-ids to form a valid, unique user file identifier, and to prefix
    program file names with mover identifiers to form valid, unique program file identifiers. The
    user should not be concerned with such requirements, and the command language does not
    permit him to form such names directly. After restructuring their arguments, these routines
    call on the file handling subprograms which follow below. The subroutines whose parameters
    are 'username, userprocess, param' are all called by the command analyzer near the label
    'short'. The command analyzer uses a fixed parameter list, hence some of these subroutines
    have unnecessary parameters.*/

definef filecom(username,userprocess,param);  /*file command*/
        /*Call the system file-definition subroutine, using arguments taken from the current
        command being interpreted. username should be a system user, and param should be a
        pair.*/
    if not ((username ∈ users) and pair(param)) then
            /*Parameters have wrong form*/
        return image + 'command rejected, improper arguments.';
    end if not;
    return file (<<username,param(1)>, param(2)>);
    end;

definef definecom(username,userprocess,param); /*program file definition command*/
        /*Interpret a 'define' command by calling the system's filedef subprogram, using argu-
        ments taken from the command being interpreted. username should be a user of the
        system, and param should be a pair.*/
```

```
    if not((username ∈ users) and pair(param)) then
        return image+' command rejected, arguments have wrong form.';
    end if not;
    return(filedef(<userprocess,param(1)>,<username,param(2)>));
    end;

definef sharecom(username,userprocess,param); /*share command*/
        /*Interpret a share command by calling the system's share routine, using arguments taken
        from the command being interpreted. username should be a system user, and param should
        be a triple.*/
    if not (username ∈ users) or type param ne tuple or #param ne 3 then
        return image+' command rejected, arguments have wrong form.';
    end if not;
    return (share(<username,param(1)>,param(2:2)));
    end;

definef permitcom(username,userprocess,param); /*permit command*/
        /*Interpret a permit command by calling the system's permit subroutine, using arguments
        from the command. username should be a system user, and param should be a triple.*/
    if not(username ∈ users) or type param ne tuple or #param ne 3 then
        return image+' command rejected, arguments have wrong form.';
    end if not;
    return (permit(<username,param(1)>,param(2),param(3)));
    end;

definef erasecom(username,userprocess,param); /*erase command*/
        /*Interpret an erase command by calling the system's erase subprogram. username should
        be a system user, and param should be a tuple of one element.*/
    if not(username ∈ users) or type param ne tuple or #param ne 1 then
        return image+' command rejected, argument has wrong form.';
    end if not;
     return erase(<username,param>);
    end;

                /*  ***************************  */
                /*            LOGOFF             */
                /*  ***************************  */

define logoffcom(u,userprocess,param);  /*logoff command*/
        /*Ignore parameters*/
    terminal write 'logoff command accepted.';
    (∀p ∈ processes{moverpart(thisprocess) is m})
        freesecondarystorage(userprocess);  /*Release swap space for process.*/
            /*Release all program file names for this user session.*/
        (∀pfn ∈ programfiles{p})
            unhook (<<m,p>,pfn>);
    end if;
        if <m,p> ne thisprocess then kill <m,p>;;
    end ∀p;
        /*Recall that the initial parameter passed to the command interpreter process was the
        device being used as a terminal by the user. That terminal is now enqueued on the logon
        process as being available for a new user.*/
    enqueue processparameter on logon;
    term;
    end;
```

```
/*  ************************************  */
/*              User-to-user messages     */
/*  ************************************  */

definef messagecom(username,userprocess,param);
      /*Interpret message command by calling the system's message subroutine. param must be
      a pair, and username and param(1) must be system users.*/
   if not((username ∈ users) and pair(param)) then
      return image+ ' command rejected, aruments have wrong form.';
   end if not;
   if param(1) ∈ users then
      return message(param(1),param(2));
   else return (param(1)+' is not a user of the system.');
   end if;
   end;

definef message(receiver,text);
   if type text ne string then
      return 'Message to be sent is not a character string.';
   end if type;
   if receiver ∈ signedon then
      if #msg(receiver) ge 10 then
            /*Reject request, too many messages already queued on receiver.*/
         return 'Request rejected. Too many messages backlogged. Try again later.';
      end if #msg;
      msg(receiver)=msg(receiver)+<owner(moverpart(thisprocess))+text>;
      return nulc;
   else return 'user not logged on';
   end if;
   end;

definef holdmsg(username,userprocess,param);
      /*Inhibit messages from being typed on the user's terminal. None of the parameters are
      used by this routine.*/
   msgon=false;
   return nulc;
   end;

definef releasemsg(username,userprocess,param);
      /*Permit messages to be typed on the user's terminal. If any are presently stacked, they
      will start to print when the next command is interpreted. None of the parameters are used
      by this routine.*/
   msgon=true;
   return nulc;
   end;


/*  ****************************  */
/*    FILE HANDLING SUBPROGRAMS   */
/*  ****************************  */

define file(par);
      /* par is of the form <fid,attr>, where fid is a file identifier of the form <uid,ufn> where
      u is a user identifier and ufn is a user file name, and where attr is a set of pairs describing
      file attributes*/
   if not(pair(par)) then
      go to badcall;
   end if not;
   fid=hd par;
   if not(pair(fid)) and fid(1) ∈ users and type fid(2) eq string) then
```

```
        go to badcall;
    end if not;
    uid=hd fid;
    if catalogue{fid} ne Ω then
        return 'file already exists, first erase old one.';
    end if catalogue;
    if ∃attr ∈ par(2) | (not(pair(attr)) or type attr(1) ne string) then
            /*attribute list has wrong format*/
        go to badcall;
    end if ∃attr;
        /*'standardfiledescription' is assumed to have been initialized to a standard default
        catalogue description of a file. Some of the items in such a description would indicate that
        the storage device is to be disc, and that the file's disposition is permanent (i.e. keep from
        session to session). cf. 3.2.3*/
    (∀attr ∈ standardfiledescription)
            /*take standard file description except if parameter is given by caller*/
        (if (∃y ∈ par(2) | hd y eq hd attr) then y else attr) in catalogue(fid);
    end ∀attr;
    allocate (fid); /*assign physical storage to file*/
    return nulc;
badcall: return image + ' command rejected, arguments have bad form.';
end file;

define filedef (pfid,ufid);
        /* pfid is a program file identifier of the form <pid,pfn> where pid is a process identifier
        and pfn is a program file name. ufid is a file identifier of the form <uid,ufn>, where uid is
        a user identifier and ufn is a user file name.*/
    if not(pair(pfid) and pfid(1) ∈ processes and type pfid(2) eq string) then
        return image + 'first parameter is not a program file i.d.';
    end if not;
    if userfile(pfid) is uf ne Ω then
            /*program file name already corresponds to a user file name.*/
        if uf eq ufid then
                /*The requested assignment has already been made. Warn the user, and otherwise
                do nothing.*/
            return 'assignment already made';
        else
                /*'pfid' had been the program file name of another file. Disassociate 'pfid' from
                that file.*/
            unhook (pfid);
        end if;
    end if;
    if catalogue (ufid) eq Ω then
        return 'the user file name is not in the catalogue. no action taken.'*/
    end if;
        /*Assignment may be made.*/
    assign(pfid, ufid, standardfixup);
    return nulc;
end filedef;


definef share(ufid,sfid);
        /*In order that file sharing may be transmitted through several levels of induction, this
        routine is coded recursively. The set done contains all the sets for which share commands
        have been issued in obeying the original share request. This is necessary to avoid endless
        recursion in case a shared library contains a reference to itself. This routine initializes done,
        and then uses the auxilliary routine, sharerec to actually do the work.*/
    global done; /*For use by inner routine, sharerec.*/
        /*Check that parameters are user file i.d.'s.*/
    if (catalogue(ufid) eq Ω) or catalogue(sfid) eq Ω then
```

```
        return image+'command rejected, some arguments are not user file i.d.s.';
    end if catalogue;
        /*initialize done*/
    done = {ufid};
        /*call on the recursive inner routine to do the real work.*/
    return sharerec(ufid, sfid);

    definef sharerec(ufid,sfid);
        /* ufid and sfid are user file identifiers. The owner of ufid wishes to use file sfid, calling it
        ufid(2), provided that sfid's owner has given permission.*/
    ownerf=hd ufid; /*owner of ufid*/
        /*Determine if ufid's owner is permitted access to file sfid.*/
    if ownerf ∈ hd[catalogue(sfid)('permit')] then
            /*Permission granted*/
            /* test if file ufid already exists.*/
        if catalogue (ufid) ne Ω then
                /*if file exists, is it shared to sfid?*/
            if catalogue(ufid)('share') eq sfid then
                return nulc; /*if yes, okay*/
            else
                    /*ufid is already defined, but not equivalenced to sfid. Warn the user but do
                    nothing.*/
                return 'file'+ufid+'already exists';
            end if;
        end if;
        if (ufid(2))(1:8) eq 'library.' and ((sfid(2))(1:8) eq 'library') then
                /*Special case for shared library. All members become shared, too. Put all shared
                file names into 'done' to avoid endless recursion.*/
            establish(sfid, tempopenclose, temp);
            temp read list;
            unhook (temp);
            (∀member ∈ (list-done))
                member in done; /*mark member as shared to avoid loop caused by circular
                    libraries. Then share members of the library, giving the caller the same file
                    names as those used by the owner of the library.*/
                sharerec (<hd ufid,member>,<hd sfid,member>);;
        end if;
        catalogue(ufid)('share')=sfid; /*Point to file being shared.*/
        catalogue(ufid)('access')=catalogue(sfid)('permit',ownerf);
        return nulc;
    else
        return 'access not permitted to file' + sfid;
    end if;
    end sharerec;
end share;

definef permit(ufid, sharers, rights);
        /*In order that permission to share files may be transmitted through several levels of
        induction, this routine is coded using a recursive auxilliary routine. The structure of this
        routine is similar to that of the share routine above.*/
    global done; /*for use in the inner routine*/
        /*Check for improper arguments.*/
    if catalogue(ufid) eq Ω /*ufid is not a user file*/
            or sharers-users ne nl /*sharers is not a subset of users*/
            or type rights ne set /*access rights have wrong format*/
            or ∃r ∈ rights | (type r ne string) then
        return image+'command rejected, arguments have wrong form.';
    end if catalogue;
        /*Initialize done to be the user file ufid.*/
    done={ufid};
```

```
        /*Call inner routine to do the work.*/
    return permitrec(ufid, sharers, rights);

definef permitrec(ufid,sharers,rights);
        /* ufid is a file identifier of the form <uid,ufn>, 'sharers' is a set of user names which are
        to be permitted access to file ufid, and 'rights' is the set of access rights which the members
        of 'sharers' have. If members of 'sharers' already have access to ufid, then we must check
        whether access rights have been reduced, in which case we must restrict access for such
        members of 'sharers', as well as to other users who were given access to ufid by such
        members of 'sharers'.*/
    uid=hd ufid; /*file owner*/
    if catalogue {ufid} eq Ω then
        return 'file does not exist';;
    if (ufid(2))(1:8) eq 'library.' then
            /*When permission is granted to access a library, give access permission to all mem-
            bers, too. All permitted file names are put in 'done' to avoid endless recursion due to
            cirular libraries (cf. 5.4.4).*/
        establish (ufid, tempopenclose, temp);
        temp read list;
        unhook (temp);
        (∀member ∈ (list-done))
            member in done;
            permitrec (<hd ufid,member>,sharers,rights);;
    end if;
    if catalogue {ufid,'permit'} is perm eq nl then
            /* The easy case. There are no former permissions outstanding for this file.*/
        catalogue(ufid)('permit')={<suid,rights>,suid ∈ sharers};
    else/*the messy case*/
        limituse(ufid,hd[perm]*sharers,rights);
    end if;
    return nulc;

define limituse(ufid,sharers,rights);
        /* ufid is a file i.d. of the form <uid,ufn>, 'sharers' is a set of user ids. All members of
        'sharers' are restricted to the access rights in the set 'rights', and the members of 'sharers',
        in turn, can only give permission to other users to use file ufid with access rights which are
        included in the set 'rights'. We omit parameter checking because this routine is called by
        'permitrec' with previously checked arguments. As an additional check on 'permitrec' and
        on the integrity of the catalogue, parameter checking could be added.*/
    (∀suid ∈ sharers)
        d=catalogue(ufid)('permit',suid); /*former access rights*/
        catalogue(ufid)('permit',suid)=rights; /*new rights*/
            /*Now look for users to whom 'suid' may have given the right to use 'ufid', and make
            sure that such users' rights don't exceed 'suid's rights. In the following statement, note
            that hd [catalogue] gives all the user-file identifiers. As coded, the next step involves a
            search, which might be very inefficient, over the whole catalogue. However, if the
            catalogue were restructured as a 2-parameter map catalogue(username, filename)
            rather than as the present catalogue(<username, filename>), this objection would be
            removed.*/
        (∀member ∈ hd [catalogue] | hd member eq suid and
                        catalogue(member)('share') eq ufid)
            limituse(member, hd {catalogue(member)('permit')},d*rights);
        end ∀member;
    end ∀suid;
end limituse;
end permitrec;

end permit;
```

```
definef erase(fid);
        /*Check that there is a cataogue entry for 'fid'*/
    if catalogue(fid) eq Ω then
        return image+'command rejected, improper argument.';
    end if catalogue;
        /*A shared file can only be erased if permission had been given to do so.*/
    if catalogue(fid)('share') ne Ω and
            not('erase' ∈ catalogue(fid)('access')) then
        return 'file cannot be erased';;
    files={fid} is handled;
    (disable)
    (1≤∀i≤#catalogue)
            /*An upper limit on the number of iterations to avoid a while-diction in a disabled
            block. This loop finds all file i.d.'s which refer to the file fid by means of sharing. The
            set 'files' is initialized to the singleton 'fid'. During each execution of the loop, the set
            'files' is augmented by all file i.d.'s which are shared with members of 'files'. Once
            'files' fails to grow as a result of this process, it contains all file identifiers which refer
            to the file being erased. Clearly, the number of iterations cannot excede the number of
            catalogue entries, and usually just one or two iterations suffice. Much the same
            comment concerning efficiency of this loop can be made as was made above, concern-
            ing a similar loop in the 'limituse' routine.*/
        files=files+({h∈hd[catalogue] | catalogue(h)('share')∈handled} – files is handled);
        if handled eq nl then
                /*We failed to find new files shared to members of 'files'*/
            quit ∀i;
        end if;
    end ∀i;
    end disable;
        /*Recall that userfile is a map from program file names to user file names (cf. section
        3.5.1.5). All program file names which refer to the file being erased are removed, in case
        running programs attempt to use this file. files is now the set of all user file identifiers
        which refer to the file being erased.*/
    (∀ufid∈files)
        (while ∃pfmap∈userfile | tl pfmap eq ufid)
                /*A program file name is defined to be the file being erased.*/
            unhook(hd pfmap); /*Program file name is now undefined*/
        end while ∃pfmap;
        if ufid ne fid then
            catalogue {ufid} = Ω; /*Delete shared entry from catalogue.*/
        end if;
    end ∀ufid;
    relinquish fid; /*Give up file's space.*/
    catalogue{fid}=Ω; /*Now drop erased file from catalogue.*/
    return nulc;
    end;

definef searching(username,userprocess,param);
        /*Interpretation of the 'searchorder' command.*/
        /*First check that param is a tuple, and 'username' is a system user.*/
    if not (type param eq tuple and username ∈ users) then
        return image+' command rejected, improper arguments.';
    end if not;
        /*Next check that all components of 'param' are strings.*/
    if ∃p(i)∈param | type p ne cstring then
        return 'Rejected, argument'+dec i+'is not a string';
    end if;
        /*All arguments have correct form.*/
        /*Next check that all libraries exist.*/
    liberror=false; /*Initialize indicator that there are missing libraries.*/
    (∀lib(i)∈param(2:))
```

```
        if lib eq '*' then continue ∀lib(i);;
        if catalogue(<username,'library.'+lib>) ne Ω then continue ∀lib(i);;
        liberror=true;
        terminal write 'library'+lib+'does not exist';
    end ∀lib(i);
    if liberror then
        return 'searchorder command rejected';
    else
        searchorder(param(1))=param(2:);
        return nulc;
    end if;
end searching;

end commandanalyzer;

                /*  ***************************  */
                /*        Get block of Main Storage        */
                /*  ***************************  */

definef qd getblock (L,P) on blockflag;
        /* L is the length of the desired block, P is the identifier of the process to get the block,
        function returns base B of the block if available, otherwise, Ω */

share secondarystorage;
        /*The above share statement declares secondarystorage to be shared among all processes
        executing this subroutine.*/

initially  /*get disk pack for swapping programs*/
    if secondarystorage ne Ω then quit initially;;
            /*This initially block is done only the first time that this routine is entered, not once
            per process.

            In the next three statements, 'swapfile' becomes the name of the system's file for
            swapping memory images between main memory and secondary storage, 'swapspace'
            becomes the program file name of that file, and 'secondarystorage' represents all the
            data on 'swapspace'.*/
    monitor('file','swapfile',{<'space','all'>});
    monitor('define','swapspace','swapfile');
    secondarystorage=open('swapspace');
end initially;

    if L lt 0 or L gt maxmem then
            /*Request is for negative space, or too much space.*/
        free blockflag;
        return Ω;
    end if;
        /*First, search for an available memory block which is large enough.*/
        if (∃s(k) ∈ memsize | s ge L and memown(k) eq Ω) then
                /*A block which is large enough has been found.*/
                reclaimed=s;
            j=k+1;
                /*At 'final', the k^th block gets assigned to process P. If the k^th block is too large,
                the remainder becomes an unassigned block which is contiguous to the k^th. All the
                blocks from the j^th to the last become contiguous to the k^th (or the k+1^st, if the k^th
                was larger than the amount requested for P.*/
            go to final;
        end if;
```

```
        /*Perhaps memory is too fragmented. The used memory now gets compacted by moving
        toward lower-numbered cells. If and when a large enough unused block is created, the
        memory request is fulfilled.*/
addup:  reclaimed = 0;
    foundsome=false;
        /*  reclaimed=amount of reclaimable memory, i.e. size of contiguous, available space,
            foundsome=true when some reclaimable memory has been found*/
        (∀blocks(i) ∈ memloc)
            if memown(i) eq Ω or memown(i) eq P then
                    /*Note that some blocks may already be assigned to P by the code at preempts-
                    torage, below. This occurs if during a previous execution of this block of code,
                    insufficient free storage had been found.*/
                reclaimed=reclaimed+memsize(i);
                if not foundsome then
                        /*This is the first free block found*/
                    k=i;
                    foundsome=true;
                end if not;
                if reclaimed ge L then
                        /*Enough memory fragments exist. Compact used storage to create a large,
                        free block.*/
                    B=memloc(k); /*first unused location*/
                    (k<∀j<i)
                        if n(memown(j) eq Ω or memown(j) eq P) then
                                /*move useful block to lower memory to create larger contiguous
                                unused block in higher memory. Note that since I/O does not go
                                explicitly via variables in a process's memory, but rather through a
                                buffer in a workqueue managed by the operating system, we dont't
                                have to first wait for I/O to finish before relocating a process's
                                storage. With a more conventional storage model, such a wait would
                                be required. Recall that memory is a vector to access all of main
                                memory directly. cf. section 5.3.*/
                            memory(B:memsize(j))=memory(memloc(j):memsize(j));
                                /*Adjust the relocation portion of the environment of the process
                                whose memory area was just moved.*/
                            hd relocate(state(memown(j))) = B;
                            memloc(k)=B;
                            memsize(k)=memsize(j);
                            memown(k)=memown(j);
                            B=B+memsize(j); /*readjust location of free storage*/
                            k=k+1;
                        end if;
                    end ∀j;
                        /*Allocate free block to the caller.*/
                    j=i+1;  /*j points to block after allocated block*/
                    go to final; /*where memown, memsize, and memloc get compressed.*/
                end if;
            end if;
        end ∀blocks(i);

preemptstorage:
        /*Search for processes owning memory blocks, but not dispatchable. Such blocks are
        members of workset{lowprio} + workset{hiprio}.*/
    (∀pr(i) ∈ memown | pr ne Ω)
            if pr ∈ workset{lowprio} or pr ∈ workset{hiprio} then
                /*pr is a process waiting to be scheduled, but not yet on the CPU's workqueue.
                Pre-empt its storage. Since the scheduler and this subprogram are both enqueued
                on 'blockgflag', the scheduler will not be able to pass any processes to the dispatch-
                er until 'getblock' terminates. Thus, in extreme cases, every dispatchable user
                process will exhaust its timeslice and get put into one of the scheduler's work
```

```
            queues, upon which this loop pre-empts its main storage! Passing the test 'L gt
            maxmem' at the beginning of this subprogram guarantees that eventually enough
            storage will be reclaimed.*/
            memown(i)=Ω;
        end if;
        /*Determine where in secondary storage to copy memory contents. The following
        conditional must be true for all user processes, and is used to set j to the appropriate
        value from which pr's secondary storage space can be computed.*/
    must=∃y(j) ∈ extstore | y eq pr;
    secondarystorage((j-1)*maxmem:memsize(i))=memory(memloc(i):memsize(i));
    if reclaimed + memsize(i) is reclaimed ge L then
            /*Enough memory has been preempted.*/
        go to addup;
    end if;
end ∀pr(i);
    /*Not enough storage has been reclaimed.*/
await #workset(hiprio)+#workset(lowprio) gt 0;
go to preemptstorage;
final:
        /*Make assignment of a block of memory of size L, beginning at B, to process P. Note
        that if the reclaimed block is greater than L in size, the remainder of the reclaimed
        block is kept as an unassigned block of memory.*/
    memown(k)=P;
    memloc(k)=B;
    memsize(k)=L;
        /*If the unused memory block just assigned is larger than required, keep the
        portion not required as an unused block and remove items between the k^th and j^th in
        the old vectors.*/
    memsize(k+1:)=if L lt reclaimed then <reclaimed-L> else nult + memsize(j:);
    memloc(k+1:)=if L lt reclaimed then <B+L> else nult + memloc(j:);
    if L lt reclaimed then
        memown(k+2:)=memown(j:);
            /*Indicate that the unused portion of the reclaimed block has now become a block
            in its own right which is available for assignment.*/
        memown(k+1)=Ω;
    else /*The reclaimed block was exactly the right size.*/
        memown(k+1:)=memown(j:);
    end if;
    free blockflag;
    return B;
nonexistent: return Ω;
end getblock;

            /*  ***************************  */
            /*            Scheduler          */
            /*  ***************************  */

    /*Processes are initially placed on workset(hiprio) by the command analyzers. The
    scheduler selects the first process in workset(hiprio), gets the corresponding memory into
    main memory, removes the process from workset(hiprio) and places it at the end of the
    CPU's workqueue. If there are no high priority processes, the same actions are repeated
    for low-priority processes. See Section 5.6.4.*/

getmore:
    if #workset(hiprio) ne 0 then
            /*Attempt to schedule potentially highly interactive processes.*/
        B=sched(hiprio);
        if B ne Ω then go to getmore;;
    end if;
        /*If we cannot schedule an interactive user, try a long-running process.*/
```

```
    B=sched(lowprio);
    if B ne Ω then go to getmore;;
    await #workset(hiprio) + #workset(lowprio) gt 0;
    go to getmore;

definef qd sched(queue) on blockflag;
        /*This routine and 'getblock' must have exclusive use of the memory assignment maps. As
        soon as the scheduler has determined whether or not the next user to run has his program
        in main memory, the 'getblock' routine is free to run*/

        p=getfirst(queue); /*first eligible user in queue*/
        if p ne Ω then
            if n(∃x(i) ∈ memown | x eq p) then
                free blockflag;
                B=getblock(size(p),p);
                if B eq Ω then return Ω;;
                hd relocate(environment(p))=B;
            else free blockflag;
            end if p;
            usednow(x)=0; /*time used in current time slice*/
            putlast(CPU,p);
        end if;
    return(p);
nonexistent: return Ω;
end;

                /*  ***************************  */
                /*          Dispatcher           */
                /*  ***************************  */

    /*cf. 2.4.3.1*/
getwork: waitcopy=waitset; /*Copy set of waiting processes*/
loop: if waitcopy ne nl then
        s from waitcopy;
        L=loctr(state(s));
        CPUcontrol=s; /*Reevaluate condition for resuming process s.*/
        if isok then
                /*'isok' is communication between s and the dispatcher.*/
            s out waitset;
            putlast(CPU,s);
        else loctr(state(s))=L;
        end if;
        go to loop;
    end if;
        /*Give priority to system processes.*/
    (disable)
        findfirst(CPU,s,privilege(s) eq 'system');
        if s ne Ω then
            remove(CPU,s);
                /*We don't want system routines to be interrupted often by the timer. In the
                following statement, 'onesecond' is a machine dependent macro which specifies the
                number of ticks in a second. (cf. 2.4.1.13)*/
            timer=clock+onesecond;
        else
                /*No system processes to dispatch -- choose a user process.*/
            s=getfirst(CPU);
            if s eq Ω then go to getwork;;
            starttime=clock; /*for interrupt handlers*/
                /*Compute when the quantum ends.*/
            timer=clock+quantum(moverpart(s))-usednow(moverpart(s));
```

```
        end if;
    end disable;
    CPUcontrol=s; /*give control to the chosen process*/
    go to getwork;

                /*  ***************************  */
                /*          Timer Interrupt      */
                /*  ***************************  */

timexpt:
(disable)
        /*'longtime' is a machine dependent macro specifying an appropriately large number of
        clock ticks (cf 2.4.1.13).*/
    timer=clock + longtime;
    if n (privilege(resume) eq 'system') then
            /*A user process has exhausted a time quantum. Determine whether additional quanta
            exist in the time slice, or whether the process should be put into a low priority queue
            (cf. 5.6.4).*/
        if nquant(moverpart(resume is mov)) lt mquant(mov) then
            nquant(mov)=nquant(mov)+1; /*No. Count quanta used.*/
            usednow(mov)=0; /*Initialize time used in new quantum to 0*/
            putlast(hiprio,resume); /*Process put at end of hi-priority queue*/
        else
                /*Time slices other than the first are at low priority.*/
            mquant(mov)=1; /*Only 1 quantum per time slice.*/
            quantum(mov)=onesecond; /*from now on*/
            putlast(lowprio,resume); /*but a long one*/
        end if;
    else
            /*System interrupted. Prepare to resume system process.*/
        putfirst(CPU,resume);
    end if;
        /*Give up control to the dispatcher.*/
    <CPUcontrol, loctr(thisprocess)> = <dispatcher,timexpt>;
    end disable;


                /*  ***************************  */
                /*        MONITOR SERVICES       */
                /*  ***************************  */

monitorxpt:
    (disable)
            /*First, keep track of time used by user*/
        if resume ∈ userprogs then
            usednow(resume) = usednow(resume) + clock - starttime;
        end if;
        fcn=hd cause;
        if fcn ∈ {'read', 'write', 'backspace', 'rewind', 'space', 'wait', 'enable', 'disable', 'release',
                'endfixup', 'fixup', 'abend'} then
                /*'inputrequest' is a label found below.*/
            loctr(osenvironment)=lookatrequest;
                /*Start a privileged process to interpret the monitor request. Attach this new
                process to the mover of the requesting process.*/
            state(<newat,newat>) is servicesprocess = osenvironment;
            split to servicesprocess (cause) for resume;
            remove(CPU,servicesprocess);
            <CPUcontrol, loctr(state(thisprocess))>=<servicesprocess,monitorxpt>;
                /*The then portion of this if-statement is meant to treat the same commands as
                were treated by the monitor services section of the uniprogramming system of
```

```
            Chapter III. The commands are handled essentially the same way as in Chapter III,
            except that in interpreting I/O requests for shared files, the access rights of the user
            must be checked.
        else
            /*Probable command. Enqueue the parameters on the command analyzer; no
            monitor services process need be split off.*/
          enqueue cause on ancestor(resume) for resume;
          <CPUcontrol,loctr(state(thisprocess))>=<dispatcher,monitorxpt>;
            /*See the code in the command analyzer, at labels 'waiting' and 'readdata', for
            recursive use of the command analyzer resulting from this monitor call.*/
        end if;
    end disable;

lookatrequest: ...
        /*Here, the remainder of the monitor interrupt handling code of the uniprogramming
        system, should be inserted. This includes all the code from the label 'lookatrequest' up to,
        but not including, the dispatcher. Also needed from the uniprogramming system are the
        operator message analyzer, external storage allocation, and file map maintenance sec-
        tions.*/
end interactivesystem;
```

Chapter VI

Summary

6.1 Classification of PSETL Extensions

PSETL represents an attempt to extend SETL for operating system description. The extentions, as described in Chapter II, proceded in two directions: the definition of global data objects, and the addition of control structures.

Fourteen global data objects were defined. Several of these, such as **interrupt**, **resume**, and **clock** correspond closely to hardware features found on most computers. Others, such as **workset**, **waitset**, **facilities**, and **state** are abstractions of objects which operating system software manipulates. These objects do not correspond closely to hardware components; rather, they correspond to higher level objects which operating systems control. These PSETL objects are sets or tuples and may be manipulated as ordinary SETL objects. Thus, algorithms involving such data structures (e.g. a search for a non-busy facility) can use the full power of SETL.

Four primitive control structures are provided. The **initially** block is extended to provide execution for initial execution of a subprogram by each disticnt process. The disabled-block provices for uninterruptable code. The **share** declaration provides for information sharing between distinct processes executing the same body of code. The special variable **CPUcontrol**, like the previously mentioned distinguished data structures, can be used in SETL expressions; when it is the object of a storage operator, the result is to switch control of the CPU from one process to another.

Several additional control operations (see section 2.4.3), such as **await**, **enqueue**, **free**, and the queued subprogram are particular to PSETL, but these can be expanded as macros in terms of the PSETL data and control extensions of sections 2.4.1 and 2.4.2. Thus, these are not primitive, although it is convenient to think of them as atomic operations, and they are used freely in this text.

The examples in Chapters III-V serve as test cases for coding in the PSETL defined by this family of extensions.

6.2 Experience with PSETL

Before commenting on my experiences using PSETL, some remarks on my previous operating system experience are appropriate. My experience has been limited to operating systems for IBM computers, but range over a wide variety of machines. The system includes an experimental operating system for the IBM 704, an experimental multiprogramming system for the IBM 7030 (STRETCH), FMS and IBSYS for the IBM 7090, an operating system to control coupled IBM 7040's and 7090's, OS/360 and CP/CMS for IBM System 360 and System 370. All these operating systems have one characteristic in common: they are coded in assembly language.

When I embarked on coding in PSETL, I was at the same time making my first attempt at using a high level language for expressing operating system algorithms. My initial attempts felt clumsy and were discarded. This probably would have been the case with any higher level language since any such language, by tending to suppress some details, would present me, the designer and implementor, with a different, generally cleaner, view of the computing system than I had been used to. SETL allows detailed coding decisions to be suppressed or deferred, but in my initial attempts, I could not overcome my previous training and thus tended to data structures and their manipulation in minute detail, and the resultant code looked too much like assembly language with semi-colons.

I do not want to leave the impression that PSETL is clumsy as a language for expressing operating systems; it was only my long experience with assembly language which acted as an initial handicap. Many stuctures which operating systems use are expressed very naturally as maps, and SETL is very well suited to using and manipulating such structures. The code in Chapter V is better SETL than the code in Chapter III, and reflects my increased comfort and ability to use PSETL.

If we examine the sample operating systems in this text, we find that approximately 10% of the lines of code involve PSETL dictions. PSETL dictions are concentrated in those portions of code which are involved with the creation and control of processes, and which concern vital interprocedural mechanisms such as interrupt handling. For many subprograms, ordinary SETL suffices. PSETL dictions fall naturally into three classes: dictions which reference special variables, macro operations which start, stop or postpone processes, and dictions which obtain exclusive control or exclusive use of data. The first group of dictions is easily incorporated into SETL -- it is as though certain objects were globally defined and available for use. The second group also poses little difficulty, for they naturally appeal when the system programmer requires a means to synchronize processes. The third group, involving exclusiveness, is the most difficult to use, for the need to exert exclusive control is not generally apparent when an algorithm is initially being coded.

These remarks are best illustrated by an example. Consider the function called 'sched' appearing in the scheduler section of code in Chapter V. This function receives as input the identifier of a workqueue of processes. If the queue is empty, the function returns Ω. Otherwise, the first process is removed from the queue. If main memory for that process is already assigned, or can be assigned, the function returns the process's identifier; otherwise it returns Ω. Straightforward code for this function is:

```
definef sched(v);
p=getfirst(v); /*remove first process from queue*/
if p ne Ω then
    if n(∃x(i)∈memown | x eq p) then
        /*if memory is not already
          assigned, try to get it.*/
        B=getblock(size(p),p);
        if B eq Ω then return Ω;;
        /*indicate new location of memory in the
          relocation portion of the environment.*/
        hd relocate(environment(p))=B;
    end if;
    return p;
end if;
end sched;
```

However, the routine as just shown will not work correctly, since other portions of the operating system which use data in common with 'sched' may cause interference. Both 'sched' and 'getblock' use the vector 'memown' and the workqueue corresponding to v (v could be either 'hiprio' or 'lowprio'). Changes to 'memown' and the possible workqueues occur in disable blocks in 'getblock'. The execution of code between

```
p=getfirst(v);
```

and

```
if n(∃x(i) ∈ memown | x eq p) then
```

cannot safely be interleaved with execution of 'getblock'. Here this can be prevented in two ways: 'sched' can be a queued function on the facility blockflag, as is 'getblock', or the code between the two statements cited above can be made part of a a disabled block if 'getblock' also uses the workqueues which concern us only in in a disabled block. In this case, the queued function is preferable since it will still allow 'sched' to run concurrently with other disabled blocks in the system, that is, it is interruptable by other processes, and in a multi-CPU configuration, it can run while another CPU is disabled. Furthermore, 'getblock' will not require the use of disabled blocks. We must only be careful to free blockflag so that 'sched' can ultimately call upon 'getblock', and so that 'sched' or 'getblock' can be reused. Hence, the final form of 'sched' is:

```
definef qd sched(v) on blockflag;
p=getfirst(v);
if p ne Ω then
    if n(∃x(i) ∈ memown | x eq p) then
        free blockflag; /*allow getblock to run*/
        B=getblock(size(p),p);
        if B eq Ω then return Ω;;
        hd relocated(environment(p))=B;
        return p;
    end if;
end if;
free blockflag; /*all paths must free blockflag*/
return p;
end sched;
```

PSETL allowed a simple, straighforward expression of the 'sched' algorithm as given above in the first rendition. After considering the possible conflicting use of data, PSETL also allowed a simple modification to get the desired exclusive control over two data structures. The expressivity of PSETL was well suited for this short example, as indeed, it has been for all the code in this text.

A PSETL shortcoming which shows in this example was that correct coding required awareness by the programmer of possible conflicting use of data. Then, with some care, the necessary PSETL dictions were used to prevent the conflict. PSETL makes it easy to write such code, but it does not go very far in removing the need for awareness of conflicting reference to common data by possibly concurrent processes. In this case, PSETL design follows contemporary computer design too closely.

The disable block of PSETL is the one instance where PSETL breaks away from current computer design with some success. The restrictions on the contents of a disable block guarantee the termination of a disable condition. Had we extended SETL by means of **enable** and **disable**

statements, then the common programming problem of assuring exit from the disabled state would exist in PSETL also. It is true that the restrictions on the disable block did at times make for cumbersome construction, but the burden is certainly no worse than that carried by devotees of go-to-less programming.

6.3 Future Directions

The system programmer would benefit from additional extensions to PSETL which would relieve him more fully from concern about interaction between concurrent processes. As PSETL now stands, data items may be treated as facilities, reserved via the queued subprogram 'reserve' (see 2.2.6.1), and released via **free**. However, these mechanisms are voluntary; failure to reserve prior to use can lead to unpredictable results, and failure to free can lead to deadlock. The deadlock problem also arises if the reserve-free mechanism is carelessly used, since two processes can request the same facilities in reverse order.

Linguistic tools which require reservation of critical data prior to use are needed. A 'reserved block' diction such as:

```
reserve v;
    block;
end reserve v;
```

might be appropriate. However, to prevent accidental failure to exit from the block, the same restrictions as we have placed on the disabled block would have to apply. Along with the reserved block, it is necessary to specify the critical variables which are only to be accessible in reserved blocks. A modification of the **own** declaration might serve this purpose, e.g.,

```
own reserved v;
```

With the modified **own** declaration, the reserved block becomes identical to Brinch-Hansen's critical region [B], in that the compiler can determine whether a variable is used outside the reserved block. However, we have been explicit about the types of statements which cannot appear within the reserved block because of our desire to have the compiler check that the block's execution will terminate.

Nested reserved blocks must still be used with care to avoid concurrent processes with the following structure;

```
reserve a;
    reserve b;
        .
        .
        .
    end;
end;

and

reserve b;
    reserve a;
        .
        .
```

```
    end;
end;
```

from becoming deadlocked [CES, Hab, Hav, Hol]. Perhaps nested reserved blocks should be barred, in favor of requiring all reserved variables to be specified on entry to the block. This avoids piecemeal allocation of facilities to a process which is a necessary condition for deadlock to arise, but has the drawback of requiring some facilities to be reserved for an unnecessarily long time.

Alternatively, we could require that **facilities** be a vector, together with a rule that a process attempting to sieze a facility x (where x **eq facilities**(i)) could only execute

x **in busy**;

if the following conditional were true:

($\forall$f $\in$ **busy*****holds**{thisprocess}, 1 $\leq$ $\exists$j $<$ i | f **eq facilities**(j)).

Similarly, we could require that for **free** to be executed,

($\forall$f $\in$ busy*holds{thisprocess}, 1 $\leq$ $\exists$j $\leq$ i | f **eq facilities**(j)).

Clearly, this would avoid circular waiting, another necessary condition for deadlock. Still another approach would make the right to use shared data available only for a limited period of time. In such an approach, the first process to access such a variable receives exclusive control of that variable for the associated time interval. During this interval, other processes attempting to reference it are automatically suspended. Failure to release the variable within the time interval results in an interrupt and loss of access to the variable to other suspended processes. If the process losing control of a timed variable is in a disabled block, the interrupt and loss of control of the variable are delayed until the process becomes enabled. This approach avoids non-preemptive scheduling, which is also a necessary condition for deadlock.

The PSETL rule making it illegal to use subroutines from within a disabled block (or the proposed reserved block) could be eased as follows: Let D be the class of subprograms which have no backward branches and which have no while-blocks. Suppose further that in the prologue of each (compiled) program, we include a means whereby a subprogram can determine whether it is in class D. Entry to class D programs could then be valid in the disabled state provided that the entry is not recursive. All other subprogram entries would be invalid and would cause a distinguished interrupt to indicate forced exit from a disabled block.

The SETL name propagation rules need strengthening. At present, if a variable is to be made known across unnested scopes, the dictions **global** and **include** must be used. Once the **global** declaration is made, there is no mechanism in SETL which prevents an **include** from being used in any scope of code. This can make the variable accessible in portions of the operating system in which the programmer of that portion of the operating system which owns the variable did not mean it to be accessible. SETL should make it possible to specify which processes, or which scopes may access data, and what kind of access is to be allowed, e.g. retrieval access or assign-

ment access. A process owning data should be able to dynamically adjust the access to that data which it permits other processes to have.

SETL, being a value-oriented language, implies inefficiencies when large structures are manipulated, because of the copying which must be done. Code in which these inefficiencies were deliberately avoided at the expense of clarity of expression is found, e.g., in the command analyzers of the code in Chapters III and V, in which several input sources can be suspended during command macro expansion. To stack file variables would imply a large copying operation; therefore, in the code, the file names were stacked, and the files themselves had to be re-opened on unstacking. For structures other than files there exist no names or pointers which can be manipulated instead of the structure itself. If PSETL is to be used for realistic expression of operating systems, pointers or reference-by-name will have to be be added to the language.

BIBLIOGRAPHY

[B] Brinch-Hansen, P., "Operating System Principles", Prentice Hall, Englewood Cliffs, N.J., 1973

[C] Codd, E. F., "Multiprogram Scheduling", Comm. ACM 3, 6, pp. 347-350, June 1960; Comm. ACM 3, 7, pp. 413-418, July 1960.

[CES] Coffman, E. G., Elphick, M. J., and Shoshani, A., "System Deadlocks", ACM Computing Surveys, 3, 2, pp. 67-78, June, 1971.

[Co] Corbato, F. J., "PL/I as a Tool for Systems Programming", Datamation, 15, 5, pp. 68-76, May, 1969.

[D] Denning, P. J., "Third Generation Computing Systems", ACM Computing Surveys, 3, 4, pp. 175-216, Dec. 1971.

[DV] Dennis, J. and Van Horn, E., "Programming Semantics for Multiprogrammed Computations", Comm. ACM 9, 3, pp. 143-155, March 1966.

[Di65] Dijkstra, E. W., "Cooperating Sequential Processes", Technological University, Eindhoven, The Netherlands 1965 (Reprinted in "Programming Languages", F. Genuys, ed., Academic Press, New York, N.Y., 1968)

[Di68] Dijkstra, E. W., "The Structure of THE Multiprogramming System", Comm. ACM 11, 5, pp. 341-346, May 1968

[DM] Donovan, J. and Madnick, S., "Operating Systems", McGraw, New York, N.Y. 1974.

[Hab] Habermann, A. N., "Prevention of System Deadlocks", Comm. ACM 12, 7, pp.373-377, July 1969.

[Hav] Havender, J. W., "Avoiding Deadlock in Multitasking Systems", IBM Systems Journal, 7, 2, pp. 74-84, 1968.

[Hoa] Hoare, C. A. R., "Towards a Theory of Parallel Programming", International Seminar on Operating System Techniques, Belfast, Northern Ireland, July-August 1971.

[Hol] Holt, R. C., "Some Deadlock Properties of Computer Systems", ACM Computing Surveys, 4, 3, pp. 179-196, Sept. 1972.

[I72] "IBM Virtual Machine Facility/370: Command Language User's Guide", IBM Publication GC20-1804, 1972

[I73] "IBM System/360 Operating System: Job Control Language Reference", IBM Publication GC28-6704, 1973

[K] Knuth, D., "The Art of Computer Programming", Vol. 1, Ch. 2, Addison-Wesley, Reading, Mass. 1969.

[MMc] Morenoff, E., and Mc Lean, J. B., "Inter-program Communications, Program String Structures, and Buffer Files", Proc. AFIPS SJCC, pp. 175-183, 1967.

[S73a] Schwartz, J. S., "On Programming, an Interim Report on the SETL Project", Installment II: The SETL Language and Examples of Its Use, Courant Institute of Mathematical Sciences, N.Y.U., New York, N.Y. 1973.

[S73b] "Reflections on P. Markstein's Newsletter on SETL Extentions for Operating System Description", SETL Newsletter 98, Courant Institute of Mathematical Sciences, N.Y.U., New York, N. Y. 1973.

[W68] Wirth, N., "PL360 - A Programming Language for the 360 Computers", J. ACM 15, 1, pp. 37-74, Jan. 1968.

[W69] Wirth, N., "On Multiprogramming, Machine Coding, and Computer Organization", Comm. ACM 12, 9, pp. 489-498, Sept. 1969.

[W71] Wirth, N., "The Programming Language Pascal", Acta Informatica 1, 1, pp.35-63, 1971.

A PRECIS OF THE SETL LANGUAGE[1]

In the present section, we summarize the principal basic features of the SETL language, as they have been defined in the preceding pages. It is hoped that this *precis* can serve as a useful brief reference.

Basic Objects: *Sets* and *atoms*; sets have atoms or sets as members. Atoms may be

Integers	Examples: 0, 2, −3
Real	Examples: 9., 0.9, 0.9E−5
Boolean strings	Examples: 1b, 0b, 77b, 00b777
Character strings	Examples: 'aeiou', 'spaces- '
Label (of statement)	Examples: label:, <label:>

Blank (created by function **newat**). Ω is special 'undefined' atom.

Subroutine. Function.

The operator **type** x returns the type of the object x.

Basic operators for atoms:

Integers:	arithmetic:	+, −, *, /, // (remainder)
	comparison:	**eq, ne, lt, gt, ge, le**
	other:	**max, min, abs**

Examples: 5//2 is 1, 3 **max** −1 is 3; **abs** −2 is 2.

Reals: Above arithmetic operations (with exception of //)
plus exponential, log, and trigonometric functions.

Booleans	logical:	**and** (or **a**), **or**, **exor**, **implies** (or **imp**), **not** (or **n**)
	logical constants:	**t** (or **true**, or 1b);
		f (or **false**, or 0b).

Character strings: conversion: **dec, oct**

Examples: **dec** '12' is 12; **oct** '12' is 10.

Strings (character or boolean):
+ (catenation), * (repetition), a(i:j), a(i:) (extraction),
# (size), **nulb**, **nulc** (empty strings).

Examples: 'a' + 'b' is 'ab'; 2*1b4 is 11001100b;
2 * 'ab' is 'abab', 'abc'(1:2) is 'ab', 'abc'(2:2) is 'bc',
'abc'(2) is 'b', # 'abc' is 3, # **nulc** is 0.

General: Any two atoms may be compared using **eq** or **ne**;

[1] Reproduced from [S73a, pp 501-509]. The underlined objects in the original text have been replaced by boldface type to agree with the typography in this work.

atom a test if a is an atom.

Basic operations for sets:

$\in$ (membership test); **nl** (empty set); **arb** (arbitrary element);

(number of elements); **eq**, **ne** (equality tests);

incs (inclusion test); **with**, **less** (addition and deletion of element);

npow(k,a) (set of all subsets of a having exactly k elements).

+ (set union), * (intersection), // (symmetric difference).

Examples: a $\in$ {a,b} is **t**, a $\in$ **nl** is **f**, **arb nl** is Ω,

arb {a,b} is either a or b, #{a,b} is 2, # **nl** is 0,

{b} **with** a is {a,b}, {a,b} **less** a is {b},

{a,b} **less** c is {a,b}, {a,b} **incs** {a} is **t**.

pow({a,b}) is {**nl**,{a},{b},{a,b}}.

npow(2,{a,b,c}) is {{a,b},{a,c},{b,c}}.

Tuples

Ordered tuples are treated as SETL objects of different type than sets -- e.g. tuples may have some components undefined.

Operations on tuples:

Tuple former: If x,y,...,z are n SETL objects then

t = <x,y,...,z> is the n-tuple with the indicated components.

#t is the number of components of t

t(i:j) is the tuple whose components, for $1 \leq k \leq j$, are t(i+k−1)

hd t is t(1)

tl t is t(2:)

+ is the concatenation operator for tuples

Examples: **hd** <a,b> is a. **tl** <a,b> is <b>, which is not the same object as b.

If t = <a,b> and τ = <a,c> then

T = t + τ = <a,b,a,c>, T(3:2) = <a,c>

Tuple components may be modified by writing

t(j) = x;

An additional component may be concatenated to t by writing

t(#t + 1) = x;

Set-Definition: by enumeration: {a,b,...,c}. Set-former:

$\{e(x_1,...,x_n),\ x_1 \in e_1,\ x_2 \in e_2(x_1),...,\ x_n \in e_n(x_1,...,x_{n-1})\ |\ c(x_1,...,x_n)\}$.

The *range restrictions* x $\in$ a(y) can have the alternate *numerical form*

$\min(y) \leq x \leq \max(y)$

when a(y) is an interval of integers.

If t is a tuple, the form x(n) $\in$ t can be used, see below,

iteration headers, for additional detail.

Optional forms include

{x $\in$ a | C(x)} equivalent to {x, x $\in$ a | C(x)}; and

{e(x), x ∈ a} equivalent to {e(x), x ∈ a | t} . Functional application (of a set of ordered paris, or a programmed, value-returning function):

f{a} is {if #p **gt** 2 then **tl** p else p(2), p ∈ f | if **type** p **ne tupl**
then **f** else (#p) **ge** 2 **and** (**hd** p **eq** a)}, i.e.
is the set of all x such that <a,x> ∈ f.
f(a) is: if #f{a} **eq** 1 then **arb** f{a} else Ω,
i.e., is the unique element of f{a}, or is undefined atom.
f[a] is the union over x ∈ a of the sets f{x}, i.e., the *image* of a under f.
More generally:
f(a,b) is g(b) and f{a,b} is g{b}, where g is f{a};
f[a,b] is the union over x ∈ a and y ∈ b of f{x,y} .
If f is a value-returning function, then
f{a,b} = {f(a,b)}, f[a] = {f(x), x ∈ a}, etc.
Constructions like f{a,[b],c}, etc. are also provided.

Compound Operator:

[**op**: x ∈ s] e(x) is $e(x_1)$ **op** $e(x_2)$ **op** ... **op** $e(x_n)$, where s is $\{x_1,...,x_n\}$.
This construction is also provided in the general form
[**op**: $x_1 \in e_1$, $x_2 \in e_2(x_1)$,..., $x_n \in e_n(x_1,...,x_{n-1})$ | $C(x_1,...,x_n)$]e(x) ,
where the range restrictions may also have the alternate
numerical form, or the form appropriate for tuples.

Examples: [**max**: x ∈ {1,3,2}](x+1) is 4,
[+: x ∈ {1,3,2}](x+1) is 9,
[+: x(n) ∈ a]x is SETL form of Σa_n,
[**op**: x ∈ **nl**] e(x) is Ω.

Quantified boolean expressions:

∃x ∈ a | C(x) ∀x ∈ a | C(x)

general form is

$\exists x_1 \in a_1, x_2 \in a_2(x_1), \forall x_3 \in a_3(x_1,x_2),...$ | $c(x_1,...,x_n)$

where the range restrictions may also have the alternate numerical form, or the form appropriate for tuples.

Evaluation of

∃x ∈ a | C(x)

sets x to first value found such that C(x) **eq t**. If no such value, x becomes Ω.

The alternate forms:

min ≤ ∃x ≤ max, max ≥ ∃x ≥ min, max ≥ ∃x > min, x(n) ∈ t, etc.

of range restrictions may be used to control order of search.

Conditional expressions:

if $bool_1$ then $expn_1$ else if $bool_2$ then $expn_2$ ... else $expn_n$.

a **orm** b abbreviates if a **ne** Ω then a else b

a **andd** b abbreviates if **n** a then **f** else b

Statements: are punctuated with semicolons.

Assignment and multiple assignment statments:

a = expn; f{exp} = expn; is the same as

f = {p ∈ f | (**hd** p) **ne** exp} + {<exp,x>, x ∈ expn};

f(exp) = expn; is the same as f{exp} = {expn};

f(a,b) = expn; f{a,b} = expn; etc. also are provided.

<a,b> = expn; is the same as a = expn(1); b = expn(2);

<a,b,...,c> = expn; <a,<b,c>,...,d> = expn; etc. are also provided.

<f(a),g{b}> = expn; is the same as f(a) = expn(1); g{b} = expn(2);

Generalized forms:

<f(a), g{b,c},..., h(d)> = expn;

<f(a), <g{b,c},h(d)>,...,k(e)> = expn; etc. also are provided.

Use of general expressions on left-hand side of assignment statements (sinister calls).

$e(x_1,...,x_n) = y$; must be a no-op if executed immediately after

$y = e(x_1,...,x_n)$; and vice-versa. The use

op op' x = y;

of a product operator on the left-hand side of an assignment expands as

t = **op'** x;

op t = y;

op' X = t;

with similar rules for multiparameter compounding. These rules allow user-defined functions to be used quite generally on the left-hand side of assignment statements. The 'left hand' significance of a function is often deducible from its standard right-hand side form, but may be varied by using specially designated code blocks which are executed only if the function is called from right-hand or left-hand position respectively. These have the respective forms:

(load block);	(execution only if function called from right-hand side of assignment)
(store x) block;	(execution only if function f called is from $f(param_1,...,param_n) = x;$).

Commonly used operators having special side effects:

expn **is** x	has same value as expn and assigns this value to x
x **in** s;	same as s = s **with** x;
x **from** s;	same as x = **arb** s; s = s **less** x;
x **out** s;	same as s = s **less** x;

Use of code blocks within expressions.

If *block* is a section of text which could be the body of a function definition, then [; block] is a valid expression which both defines and calls this function. Such code block expres-

sions can be used freely within other expressions.

Control statements:

go to label;

if $cond_1$ then $block_1$ else if $cond_2$ then $block_2$ ... else $block_n$;

if $cond_1$ then $block_1$ else ... else if $cond_n$ then $block_n$;

Iteration headers:

(while cond) block;

(while cond doing blocka) block; is equivalent to (while cond) block blocka;

$(\forall x_1 \in a_1, x_2 \in a_2(x_1),..., x_n \in a_n(x_1,...,x_{n-1}) \mid C(x_1,...,x_n))$ block;

In this last form, the range restriction may have such alternate numerical forms as

$$min \leq x \leq max, \quad max \geq x \geq min, \quad min \leq x < max, \text{ etc.,}$$

which control the iteration order.

If t is a tuple, bit string or character string, then the operator of the form

$(\forall x(n) \in t)$ block; is available. This is an abbreviation for

$$(1 \leq \forall n \leq \#t \mid t(n) \textbf{ ne } \Omega)x = t(n); \text{ block;}$$

Iterators of this form may also be used in set formers,
compound operators, quantifiers, etc.

Iterator Scopes

The scope of an operator or of an *else* or *then* block may be indicated either with a semicolon, with parentheses, or in one of the following forms:

end ∀; end while; end else; end if; etc.;

or: end ∀x; end while x; end if x; etc.

Loop Control

quit; quit ∀x; quit while; quit while x;

and

continue; continue ∀x; continue while; continue while x;

The *quit* statement terminates an iteration; the *continue* statement begins the next cycle of an iteration.

Subroutines and functions (are always recursive)

To call subroutine:

sub($param_1$,...,$param_2$);

sub[a]; is equivalent to $(\forall x \in a)$ sub(x);;

Generalized forms:

sub($param_1$,[$param_2$,$param_3$],...,$param_k$)

are also provided.

To define subroutines and functions:

subroutine:

define sub(a,b,c); text end sub;

return; -- used for subroutine return

function:

definef fun (a,b,c); text end fun;

return val; -- used for function return

infix and prefix forms:

define a **infsub** b; text end **infsub**;

definef a **infin** b; text end **infin**;

define **prefsub** a; text end **prefsub**;

definef **prefun** a; text end **prefun**;

Namescopes

Scope declarations divide a SETL text into a nested collection of scopes. Scope names are known in immediately adjacent, containing, and contained scopes. Other than this, names are local to the scope in which they occur, unless propagated by **include** or **global** statements.

Declaration forms

scope name; ... ; end name;

scopes with specified numerical level

scope n name; ... ; end name;

global declaration

global $name_1$, ..., $name_n$;

with specified numerical level

global n $names_1$, ... , $name_n$;

include statement

include $list_1$, ... , $list_n$;

Example:

include bigscope1(scope1 x, scope2(-z), scope3(x, y, u[v])), bigscope2*;

'*' signifies all elements known in scope, '−' signifies exclusion of those elements listed, [ ] modifies the 'alias' under which an element is known in scope in which included. Subroutines and functions are scopes of level 0. Macros (see below) are transmitted between scopes in much the same way as variable names. The declaration

owns $routname_1(x_1, ... ,x_{n1})$, $routname_2(y_1, ... ,y_{n2})$, ...

states that the variables x_j are stacked when *routname*$_1$ is entered recursively, the variables y_j are stacked when *routname*$_2$ is entered recursively, etc.

Macro blocks

To define a block: **macro** mac(a,b); text **endm** mac;

to use: mac(c,d);

Initialization

initially block; (*block* executed only first time process entered)

Input-Output

Unformatted character string:

A SETL file is a pair <st,n> where st is a character string and n an integer indexing one of its characters.

er is end record character; **input**, **output** are standard I/O media;

the function record(s); -- reads a file <st,n> from position n

till **er** character or string-end is encountered in the character

string *st*.

Standard format I/O

An interval file f in SETL is a pair <st,n> consisting of a character string *st* and an index n to one of the characters of *st*.

f **read** $name_1$, ... ,$name_n$; using standard format reads from file <st,n>, starting at position n

f **print** $expn_1$, ... ,$expn_n$; using standard form transfers external representation of objects to file s = <st,n>, starting at position n as above[2].

The set $\{s_1, \ldots, s_n\}$ is represented as $\{r_1, \ldots, r_n\}$, where r_j is the external representation of s_j. Similarly, the tuple $<s_1, \ldots, s_n>$ is represented as $<r_1, \ldots, r_n>$.

An external file *st* in SETL is character string catalogued with the operating system supporting SETL under some identifying name *catname* (which is itself a string). The statement

x = **open** catname;

makes the string *st* into the value of x. The call

close(st,catname);

makes the SETL string *st* into the contents of the external file named by the string *catname*.

[2]In this text, we have used **write** instead of **print**, since *print* is often interpreted as meaning outputting to a typewriter terminal or a high speed printer. We want our verb to denote a general output action, and have therefore chosen the word *write*.