SETL Extensions for Operating System Description

P. Markstein

Installment I of a Thesis Draft

ABSTRACT

During the past decade, operating systems have come to play an increasingly important role in computing, to the point where today, an operating system is considered to be an integral part of a computing system. This dissertation attempts to distill diverse efforts in operating systems design, and to depict those fundemental algorithms which are peculiar to operating systems.

Our first task will be to make the notion 'operating system' as precise as possible. This will not be done with the conciseness and precision of a definition occuring in a mathematical text, but by setting forth the objectives to be satisfied by the programs to be considered. Examination of these objectives will lead to a coarse characterization of operating system algorithms.

Use of an appropriate programming language will be necessary if the algorithmic content of operating systems is to be presented in a satisfactory manner. To keep the algorithms at a sufficiently high level (uncluttered by details imposed by the language), the algorithms will be described in PSETL, a version of SETL which has been enlarged to accomodate algorithms involving interrupts, parallelism, and to some extent, machine dependent features.

Using PSETL, several operating systems will be presented in detail. The first, a simple uniprogrammed batch system, illustrates basic control mechanisms and scheduling. The second, a multiprogrammed system, introduces additional complexities due to contention for resources and conflicting objectives. Of course, the design of these systems will involve several ad hoc decisions; the reasons for the strategies adopted, as well as viable alternatives, will be discussed in the final chapter.

A detailed discussion of the major components and problem areas of operating systems includes the following topics: system nucleus, scheduling, resource allocation, control of processes, data management, virtual machines, measurements, and operating system developement and maintenance. Key algorithms are identified and presented for these areas. When alternative approaches exist, these are presented with appraisals of their relative merits and weaknesses.

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 what will be considered an operating system in this work will be given, motivated by citing objectives which such systems attempt to satisfy.

1.1 Objectives of Operating Systems

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 occurance of various machine halts. At the end of a run, computer memory was generally cleared, tape reels associated wit the concluded run dismounted, and tape reels for the next run mounted. Transition time between runs was frequently on 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, 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 characteristics information could 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 will be called 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 uses relations between the output of one step and the input to the next. In short, by requiring job or step transition information and resource requirements to be stated precisely, a computer program could 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 faithfully reaching the operating system at the conclusion of a job or step. This is insured either through software conventions, hardware, or both. Modern computers have hardware facilities which can be employed to guarantee integrety of the operating system, and to enforce its software conventions.

The ability to make a computer progress smoothly from one job step to the next is one of the most important characteristics of operating systems. Historically, operating systems came into being as a result of reducing job transition time by the use of computers.

1.1.2 Program Libraries

For any computer, 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 applications such as sorting. Rather than require each user to supply his own copy of such programs, a computer installation maintains a library of these frequently used programs, and the 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 program.

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 classes of subsets can be easily extracted. In its most primitve form, data management merely subsumes some of the complexities of coding input-output instructions; in its advanced forms, data management provides 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 prevent excessive system idle time between runs. This function of operating systems, while perhaps 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 contain more equipment than any single job in the installation can use. The motivation for such configurations is to be able to offer a wide class of services. For example, large accounting systems might require many tape or disc drives but not much main memory, while even moderate linear programming problems can profit from large main memory. 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 that 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 undesireable 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 of programs 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 return control to the task which had just completed it's idle time.

The sharing of hardware resources by independent jobs in the manner described is called multiprogramming. To successfully multiprogram a computing system, the operating system may require characterizations of the jobs being submitted for execution. Such information can be supplied through the job control language. Assuming a surplus of work, a possible objective of an operating system in scheduling jobs and determinging which jobs are to run concurrently is to minimze the rental paid for idle equipment.

In practice, however, the objective function to be minimized is subject to various constraints, such as job deadlines. 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 interactive computing or time-sharing, which on the surface appears to require excessive hardware, but, when including the human factor, may be economically justifiable.

Lower cost is not sufficient for users to agree to run on a multiprogrammed computer. They must also 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, and 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

In the discussion of automatic operation and resource utilization, the need was mentioned for the operating system to guarantee the integrity of programs sharing a computing system. This is achieved by removing from the users direct control of some hardware features, and making

those features available only through simulation, during which misuse can be detected and prevented.

The hardware features which 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, setting of boundaries in main memory such that an attempt to reference beyond the boundary creates an exceptional condition, an interrupt system, and means to place the computer in either a problem state or 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 of several users, and the operating system is required to make the correct correspondences between the various users and records on the physical device. Thus even if hardware were able to restrict a program to access only certain I/O devices, this would not offer adequate protection. To replace the privileged I/O instructions, the operating system provides routines which users may invoke corresponding to each of the hardware I/O instructions. The casual user may even derive a benefit if the operating system provides additional facilities which automatically provide buffering and synchronization between I/O and computing.

Control of physical I/O addresses is often reserved to the operating system. The rationale for removing this level of control from the user follows from the desire to maximize the chance that arbitrary programs can be multiprogrammed; for if two programs depended on using the same physical I/O unit from a set of identical devices, these two prgorams could never run concurrently. Job control language provides mechanisms to make 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. Generally, an interrupt system is simulated for user programs, so that these programs can also treat exceptional conditions without executing tests which usually fail. On the other hand, many programs are relieved of any necessity to concern themselves with interrupts.

To allow sharing of main memory, users state how much contiguous space is required, but usually do not have the freedom to specify the actual addresses in memory where the space will be. This limitation provides minimal user discomfort, since the use of relocating loaders has already preempted some control over memory allocation. However, computing systems with "virtual memory capabilities" 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 arbitrary memory locations, at his disposal, and with even more memory than is physically present.

1.1.5 Preparation and Maintenance of Software

Having stated some objectives for operating systems, one is ultimately faced with producing a set of programs, which is the operating system itself. Several questions present themselves: What shall be the environment for initially designing, developing and debugging the operating system? What features shall be introduced into the operating system for self maintenance: what instrumentation, software error detection, what ability to test the operating system under itself? Flexibility of the operating system to accomodate a wide variety of hardware configurations is also an important design issue.

Not every computing system is suitable for the construction of operating systems or for their maintenance. However, a significant fraction of an operating system consists of library programs which behave as user programs and which are thus maintained as ordinary user programs. Language processors are examples of such components of operating systems. 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 the compiled operating system programs into a new operating system. The portions of the operating system which are difficult to debug in an operating system environment are those routines involved with hardware and resource allocation.

1.2 Overview of Operating System Internals

From the operating system objectives, some of the structure and unique characteristics of these programs can be deduced. 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 for unusual occurances. 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 at these times 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 these 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 in its simplest designs, an 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 burst of CPU usage between which only monitoring of unusual events is required. Examples of functions having this property 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. Functions such as these 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 programs 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 such a form that any of the dormant programs may be restarted.

The detailed description of the state of a computation is machine dependent; however, it can be characterized well enough without such details. The state of an interrupted computation consists of all the information necessary to resume the computation. This information falls into three broad categories: data resident in the registers of the CPU, data resident in the address space of the computation, and data resident in files.

For each program sharing the CPU, the operating system reserves a portion of memory addressable only by the operating system for storing the CPU resident data while the program does not control the CPU. Generally computers can store or restore CPU resident data with only a few instructions. One such datum is the location at which to resume execution, and this is the last datum which the operating system restores when returning control to a program.

In many batch system, address space resident data are memory resident for the program's entire run, including times when the program does not control the CPU. An alternative is to copy memory resident data onto a file accessible only by the operating system when a program loses control of the CPU, and to restore it to memory before returning CPU control to the program. Other schemes involve maintaining only a fraction of address space resident data in random access memory; this approach will be discussed in greater detail under Virtual Memory.

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

1.2.2 Mappings

It has already been observed that programs in an operating system environment cannot directly access many of the computer's resources for reasons of security, and uncertainties over which of several identical resources will be assigned. Consequently, such resources are referenced by programmer invented symbols, rather than physical adresses. The operating system, during the scheduling process assigns real resources to the symbolically named resources, and creates a map from symbolic name space to the real resources. Operations on symbolically named devices are interpreted, and during the interpretation process the symbolic names are mapped into resource

addresses. Explicit inverse maps or the ability to compute the inverse must also be available in order to correlate signals from real devices with the symbolically named devices.

Maps between various symbolic name spaces and device address spaces can consume a large fraction of the space occupied by operating systems, and many system actions employ these maps or their inverses. Often composite mappings are computed. For example, in file manipulation, the following spaces are involved: external file name space, external volume name space, symbolic file name space, and physical device address space.

Chapter II
Parallel SETL

2.0 Language Requirements

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

The higher level language to be used should have the property of not forcing artificial structure on the data which the operating system manipulates. Indeed, in practice, the structure chosen may have a great bearing on performance, but this choice of data structure may be hardware dependent and should not be dictated by the language chosen. In the programs to be given here, the focus will be on algorithmic content. The desire for a structure free notation will become apparent when the many maps which an operating system requires are considered. In using these maps, a crisp, mathematical notation preserves the spirit of the algorithm, which would otherwise be obscured by structural manipulation.

SETL satisfies the requirement for structure independence and all algorithms, after appropriate discussion, will be presented as SETL programs. It will be particularly advantageous to have arbitrary index sets without explicit attention to how an index set maps into the integers or other preferred entities.

2.1 SETL Deficiencies

There are several notions which distinguish operating systems which cannot be expressed in SETL (or other commonly available higher level languages). Operating systems utilize multiprogramming, and mechanisms are required to identify the several processes comprising the operating system, and to specify the passing of control between processes. An interrupt mechanism is necessary as a means of communicating between operating system and user programs, and it must be possible to specify protection mechanisms in order to have concurrent programs with safety. Other features which must be described in algorithms, but which are inaccessible in standard SETL, include clocks and timers, external device communication, resource allocation, and resource sharing.

Idealized versions of these features will be added to SETL to make it possible to describe operating system algorithms. As with other features of higher level languages, the operating system extensions will not necessarily correspond directly to the hardware of a specific machine, but these extensions can all be realized on third generation or later computing systems. SETL with operating system extensions will be called PSETL, short for parallel SETL.

Enhancements to SETL will take several forms. Special sets will be defined within SETL to indicate the state of components of the computing system, such as the process in control of the CPU. These sets will be accessible to select operating system routines but not to user programs. As a matter of notation, names of special sets, which are only accessible by the operating system, will be underlined, both

in this text and in programs.

New operations will be defined, although 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 of these "macros" as fundamental operations. Of course, the representation of these macros in terms of a stripped down PSETL embodies some of the most fundamental operating system algorithms, and these will be described in great detail in Chapter IV.

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 names of all currently active jobs.

In discussing the operating system's handling of a job, it is not sufficient to take into account only the code (i.e. program) and data which comprise the job; the execution of the program must also be considered. The words 'program' and 'procedure' will be reserved to mean the (static) pattern of bits which the hardware is 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 sharpened 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. 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 identical with the history of a CPU's running of a program. In order to represent 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 as, 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 forcing the CPU 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 resuming at 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, process, contains elements of the form <m,p>, where m is a mover and p a

process belonging to m. The set process(m) consists of all active processes belonging to the mover m.

Let us first consider the case in which only a single CPU is present. The special set state defines the process controlling the CPU. We shall make use of three positional macros which isolate the components of state: processpart, environment, and privilege. processpart(state) is a member of process and identifies the mover and process currently controlling the CPU. environment(state) gives all the information necessary to define the path which execution of a process will take when the process comes into control of the CPU. This includes 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. privilege(state) identifies whether or not the process controlling the CPU may issue privileged operations. For example, only a privileged process may change state.

Process switching is achieved by changing state. (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 state.) 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)=L; ,

where we assume that loctr extracts the component of environment(state) which defines the next instruction to be executed within the current subroutine, 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.

Two features are required to describe an interrupt system. It must be possible to describe which portion of code 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,place>, where int specifies an interrupt class, and 'place' identifies the portion of code invoked when an interrupt of class int is encountered. 'place' must be of an appropriate form to specify a process state, as described in 2.2.1. A set resume takes on the value which state had immediately before the moment of interruption, and can be used to resume the interrupted process, via the simple statement:

state = resume;

Further details concerning an interrupt are contained in the variable cause; the value of this variable is modified to show all relevant interrupt-related information whenever an interrupt occurs. Only privileged operating system code has access to the sets interrupt, cause, and resume, which correspond to the programmable hardware mechanism which determines to where control flows after an interrupt.

In PSETL, the interrupt system is generally active or enabled, so that

interruption is generally possible. In certain routines, 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 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 quaranteed uninterrupted control. The restrictions on the disabled block quarantee 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 quarantee integrity of special sets during their use.

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 'seperate' 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 v1,v2,...,vn;
```

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 through a special set, workset, and mechanisms for adding and deleting elements of queues. For an object x, workset(x) is the queue of work stacked on x. The structure of the queues is immaterial to most of our discussion; suffice it to say that either linked lists or tuples can serve as an appropriate structure.

Two subprograms, which define the structure of workset as a queue, are provided with workset. The function getfirst(x) returns the first item in the workset for x and removes it from the workset. The routine putlast(x,y) adds y to the end of the workset for x. Both getfirst and putlast operate disabled.

Various central 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 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∈process, then holds(p) is that subset of busy which is dedicated to p.

We also regard the pool of a available CPUs as an object with a workset. The workset associated with the pool of CPUs contains the states for all 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):

```
state = getfirst(CPU);
```

To ease the coding of the common operating system operation of delaying 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(a1,...,an) 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 a1,...,an.

Each queued subroutine must use the label "nonexistent" in its body, to which control passes in the event that 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. 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);

is used to begin a new process from the state s; the process is identified by processpart(s), and execution begins at loctr(s). The pair <p1,e> is passed to this process through its environment, where p1 identifies the process which issued the split. The new process can extract the pair <p1,e> from its environment by applying the positional macro initialvar to its state. Moreover, the positional macros 'ancestor' and 'info' retrieve p1 and e from initialvar(s). Thus, a process may identify the process which initiated it by retrieving ancestor(initialvar(state)), and it may reference the information being passed to it by retrieving info(initialvar(state)).

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 have their states saved in the special set waitset. When a process x is entered into waitset, loctr(x) is set up it 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 p2;
```

enters the pair <p1,e> on p2's workqueue; here p1=processpart(state), state being the state of the process executing the request. The process p2 must be written to examine its workqueue for additional requests at the conclusion of servicing each 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 p2;
```

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 can be called to delay a process until a facility x can be secured:

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

2.2.6.2 Dijkstra 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 in 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 process 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 V;
```

Clearly, if one merely desires to synchronize processes, without requiring that a count of delayed processes be kept explicitly for 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 sieze 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; loctr(x)=S;
(∀fac∈d) processpart(x)=<w,newat>;
   split to x(fac); end ∀fac;
await #process{w}eq 0;
   .
   .
   .
S: reserve(info(getfirst(processpart(state) is q) is h) is fac));
(disable) h in holds;
          <q,fac> out holds;
          term; end disable;
```

Recall that getfirst(processpart(state)) 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;

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≤Ek≤#str|str(k) eq er) doing j=k+1;)
      printout str(j:k-j);
   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.

2.4 Detailed Summary 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 Chapter 4, alternative expansions for some of these macros will be considered.

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

fac represents a facility,
p,p1,p2,... represent elements of process,
m represents an element of movers,
n identifies a path of control for a mover,
s,s1,... represent states,
i,i1,... represent interrupt classes,
j,j1,... represent system objects having worksets,
a1,...,an represent arguments to subprograms or processes,
v1,...vn 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 process

Elements: pairs of the form <m,n> where m∈movers, and n is a path of control for the mover n.
Uses: For m∈movers, process(m) identifies the set of paths of control associated with the mover m. The elements of process are the processes in progress under control of the operating system.

2.4.1.3 state

Uses: state identifies the process currently controlling the CPU, its privilege class, and its environment, i.e. all that information concerning 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. These components of state are retrieved by the positional macros processpart, privilege, and environment, respectively. A running privileged process can always identify itself by w=processpart(state); and then refer to its workqueue, if appropriate, by workset(w) or getfirst(w).

2.4.1.4 interrupt

Elements: pairs of the form <i,s>
Uses: If the pair <i,s>∈interrupt, and an interrupt of class i occurs, then state is set to s, the former contents of state saved in resume, and the variable cause comes to represent all relevant information concerning the interrupt that has just occurred.

2.4.1.5 resume

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

```
state=resume;
```

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

2.4.1.6 cause

Elements: machine dependent
Uses: Makes available, in some suitable form, an abstract '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.

2.4.1.7 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 the detailed structure of the queue workset(j) is not needed in the algorithms, since the subroutines getfirst, remove, and putlast perform the queue manipulations by using auxilliary 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.

2.4.1.8 waitset

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

2.4.1.9 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.

2.4.1.10 busy

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

2.4.1.11 holds

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

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 others. 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 'state'

A CPU is directed to switch processes by assignment to the special

variable state. The assignment state=s; causes the process processpart(s) to control the CPU starting at loctr(s), and to operate in the privileqe class privileqe(s).

2.4.2.4 shared variables

Statement form: share v1,v2,...,vn;
Description: Variables declared in a share statement and owned by a subproqram have only one instance in storage, regardless of the number of distinct processes executing the subprogram. Such variables, especially if qlobal, may be used to communicate between subprocesses. For variables not declared as being shared, a unique value exists for each process executing the subprogram.

2.4.3 Macro Operations

2.4.3.1 await

Statement form: await cond;
Description: The privileged process issuinq 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) s=state; loctr(s)=L; s in waitset;
        go to getwork; end disable;
     L: (disable) <isok,locgoto>=<cond,M>;
           go to sortout; end disable;
      end if;
   M:
```

In this expansion, L and M are compiler generated labels, and s a compiler qenerated variable. The expansion works in cooperation with the operating system's dispatcher, getwork. 'getwork' and 'sortout' are labels in the dispatcher. The variables 'isok' and 'locqoto' transmit to the dispatcher the recomputed condition and the location from which the process resumes computation when the condition is satisfied. A simple dispatcher is:

```
   getwork: waitcopy=waitset;
   loop: if waitcopy ne nl then
         s from waitcopy;
         state=s; /*recompute condition*/
      sortout: if isok then s out waitcopy;
            loctr(s)=locqoto;
            putlast(CPU,s); end if;
         qo to loop; end if;
      /*if waitcopy eq nl then*/
      s=qetfirst(CPU);
      if s eq om then go to getwork;;
      state=s; /*qive control of CPU to chosen process*/
```

Notice in the fourth line that executinq state=s; causes control to flow to loctr(s), which corresponds to the label L in the above expansion for await. The dispatcher first tests all awaited conditions, and moves processes with satisfied conditions to the CPU's workqueue. The first element of the CPU's workqueue is then 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.

2.4.3.2 queued subprogram header

Statement forms: (1) define qd name(a1,...,an) on fac;
(2) definef qd name(a1,...,an) 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, fac∈busy and <processpart(state),fac>∈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 fac∉facilities then go to nonexistent;;
(disable) if fac∈busy then
      save=state; loctr(save)=M;
      putlast(fac,save); go to getwork;
   else fac in busy; <processpart(state),fac> in holds;;
   end disable;
M:
```

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);
   <processpart(state),fac> out holds;
   if v=om then v out busy;
      else putlast(CPU,v);
         <processpart(v),fac> in holds;; end disable;
```

2.4.3.4 split

Statement form: split to s(e);

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 process, the pair <p1,e> is stored in p's environment in such a manner that it can be retrieved by initialvar(s), (where p1 represents the process which executed the split), and and an entry is made on the CPU's workqueue, indicating that the process p begins its execution at loctr(s).

Expansion:

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

2.4.3.5 enqueue

Statement form: enqueue e on p;

Description: The pair <p1,e> is placed at the end of p's workqueue, where p1 is the process issuing the enqueue statement. Upon adding the pair to p's workqueue, p1 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,<p1,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 processpart(state); and is used by a process to terminate its own execution.

Expansion:

```
(disable)
   (∀x∈hd[workset], ∀y∈workset{x})
      if ancestor(y) eq p then remove(x,y);;;
   (∀fac∈holds{p}) free fac;;
   workset{p}=om;
   if #process{moverpart(p)} eq 0 then
      moverpart(p) out movers;;
   if processpart(state) eq p then go to getwork;;
   end disable;
```

2.4.3.7 queue management subprograms

Statement forms: getfirst(j);
putlast(j,x)
remove(j,x)

Description: getfirst(j) returns the first element on object j's workqueue and removes that element from the queue. If there are no elements in j's workqueue, then getfirst(j)=om putlast(j,x) stores x as the last element on j's workqueue. 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.

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 might 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(s)
loctr(s)
initialvar(s)
ancestor(x)
info(x)
moverpart(p)

Description: Once a specific structure for state has been chosen, the first three macros, to be used on sets with the same structure as state, extract the process portion, environment portion, and privilege class portion of their arguments, respectively. If for example, state is a triple, we could use the conventions processpart(s)=s(1), environment(s)=s(2), and privilege(s)=s(3).

loctr and initialvar also apply to objects with the same structure as state. loctr(s) extracts the location portion of s's environment, and initialvar(s) extracts the pair <p,e> from environment(s), where p is the process which initiated processpart(s), and e is initialization information passed by p to processpart(s).

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.