SYNCHRONIZATION AND SIMULATION IN OPERATING SYSTEM CONSTRUCTION

71-102

A Thesis

Presented to the Faculty of the Graduate School of Cornell University for the Degree of Doctor of Philosophy

by

Nelson Herbert Weiderman

September 1971

BIOGRAPHICAL SKETCH

Nelson Herbert Weiderman was born in Jersey City, New Jersey on August 13, 1945. In 1967 he received a Bachelor of Arts degree in Mathematics from Cornell University. He began graduate study at Cornell in the Department of Computer Science in September 1967. He is a member of Phi Eta Sigma, Phi Kappa Phi, and the Association for Computing Machinery. The author is married and has one child.

In memory of my father

Herbert Lawrence Weiderman, P.P.M.

whose unfailing sense of humor

lightened many lives

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to Professor Alan C. Shaw for his guidance and encouragement throughout the preparation of this thesis. He would also like to thank Professors R.J. Walker and John Hopcroft for serving on his special committee.

Many fellow students, particularly Ric Holt and Tom Wilcox, read early versions of the research presented here and provided helpful comments and suggestions.

This research was supported in part by National Science Foundation Grants GJ-108 and GJ-28176.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

This thesis is concerned with the design and implementation of computer operating systems. More specifically it deals with the problem of synchronizing many activities within the computer system for the purposes of resource allocation, communication, and data transfer. It also describes a method of simulating an operating system such that the simulation program can be used as a basis for the operating system implementation. Finally, it presents a methodology for constructing operating systems based on the new ideas for synchronization and simulation.

An <u>operating system</u> is defined as the software required to transform the hardware of a computer into an efficient, user-oriented abstract machine. The software of an operating system consists of two distinct parts called the control program and the service programs. The service programs comprise language translators, loaders, and utilities which provide a variety of services to user jobs in the computer system. The control program, consisting of schedulers, interrupt handlers, resource allocators, file systems, and input/output (I/O) services, has as one of its main functions the coordination of many concurrent activities within the computer system. Often the term "operating system" is used to refer only to the control program and is used in this sense in the rest of the thesis.

Although there is no standardized terminology in the area of operating systems, the following definitions will be helpful in future discussions. A process is loosely defined as an instance of execution of a program. A multiprogrammed or multiprogramming system is one in which the resources of the computer are being shared among many systems and user processes at the same time. This is accomplished by a rapid switching (multiplexing) among active processes.

A multiprogramming system usually operates in one of two modes. In a batch system the user submits a job which is not returned until processing of the job is completed. A time sharing system allows users to interact with their program at a keyboard or display device during its execution. Both batch and time sharing systems may have more than one user job occupying main storage simultaneously.

Most multiprogrammed batch processing systems are SPOOLed (for Simultaneous Peripheral Operations On Line). This means that prior to execution a user job is read from the input device to an auxiliary storage device. After execution the job is written from the auxiliary storage device to the output device. This permits the execution of many user jobs to be interleaved because of random access to input and output files. Also, the I/O operations can be performed more rapidly between main storage and auxiliary storage than between main storage and the sequential input or output devices, so the total elapsed time in the execution phase is reduced.

1.1 The state of the art

The state of the art in operating system design and construction is well defined by the proceedings of four conferences. The First and Second Symposium on Operating Systems Principles held in Gatlinburg, Tennessee in October 1967 [ACM 67] and in Princeton, New Jersey in October 1969 [ACM 69] are devoted to a wide spectrum of work specifically concerned with operating systems. Two conferences sponsored by NATO were held in Brussels in October 1968 [Nau 69] and in Rome in October 1969 [Bux 70]. The NATO conferences dealt with software engineering problems, but concentrated on large software projects such as operating systems.

Two points continually reappear throughout the papers presented in these four conferences. First they stress that software engineering, or the planning, design, and construction of software, is still in the rudimentary stages of its development and that we have failed in general to produce well engineered software. This was clearly the sentiment of R.M. Graham when he said [Nau 69]:

> We build systems like the Wright brothers built airplanes - build the whole thing, push it off the cliff, let it crash, and start over again.

The primitive state of software design is evident from the huge investments needed to build systems which do not behave as efficiently or as reliability as expected. Probably the best example of the large expenditures required is OS/360 which is reputed to have cost 50 million dollars and 5000 ma years to develop [Nau 69].

The second point that is stressed throughout the literature is that in order to build better software we must structure the designs so that we can construct the system in small and manageable steps. The key idea here is that systems should be evolutionary in nature. Much of the effort in this area has been devoted to deciding the best way to partition a system for a "divide-and-conquer" approach.

A third point concerns the role of simulation in the design and construction of operating systems. It is apparent that simulation has in general failed to be a useful tool in operating systems development. It has failed for three reasons:

(1) Simulations which are faithful to the system they model are often as costly and time consuming as the system itself.

(2) The validity of simulations can always be called into question and is difficult to verify.

(3) Since a simulation must perform many of the algorithms that the system performs, there is a costly duplication of effort in developing two similar software systems.

N.R. Nielsen [Nie 67] reports lack of success by three major computer manufacturers to simulate new systems for their machines because of the problems cited above.

On the positive side, Dijkstra's work on the THE multiprogramming system [Dij 68] stands out as an important contribution to the state of the art. This implementation represents a methodical and evolutionary approach to the development of reliable software. Dijkstra has also made important contributions in the areas of synchronization primitives for process control [Dij 65], and programming style [Dij 69].

The problem of making simulation a useful tool in operating system construction has been studied by Zurcher and Randell [Zur 68]. This is also an evolutionary approach in which the design is simulated at each of several levels with the simulation providing feedback on anticipated performance.

Another area that has been the object of study is the design of structures and operations which provide a conceptual framework for convenient solution of operating systems problems. Work in this area has been largely concentrated on process control and resource allocation. Brinch Hansen [Bri 69] has developed the RC 4000 operating system from a set of primitive operations called the nucleus.

Many of the ideas presented in this thesis are extentions or refinements of previous investigations. Where this is the case, more detailed explanations of the work will be given.

1.2 Contributions

This thesis makes contributions in three areas:

(1) process synchronization and resource allocation

(2) simulation of operating systems

(3) methodologies for operating system construction

The first novel idea is concerned with the synchronization of the processes within a computer system. Processes must interact in a computer system for a number of reasons. Their logical progress is stopped and restarted many times. Processes may become logically blocked waiting for system resources, waiting for the completion of an I/O operation, waiting for an empty or full buffer, waiting for a signal from another process, etc.

Previously each of these various synchronization problems was handled by primitive synchronization operations augmented by the structures and algorithms appropriate for a particular problem. This results in a number of *ad hoc* mechanisms for problems which are essentially similar in structure.

The thesis presents a more unified approach to the problem of process synchronization by defining a resource as anything which may cause a process to become blocked. The features common to all synchronization problems are extracted into two primitives designated REQUEST and RELEASE which operate on a common basic data structure called a resource semaphore. A process may become logically blocked only by invoking a REQUEST and can only be reactivated when another process executes a RELEASE. Algorithms for deciding which of a number of processes is to be reactivated or which of a number of resources is to be allocated are invoked by REQUEST and RELEASE, but are left variable.

By introducing these primitives for process synchronization and resource allocation many of the tedious details which would normally be implemented in many places in the operating system are suppressed. The restriction to two primitives for all process synchronization also makes possible the systematic analysis of an operating system for the purpose of detection, prevention, and recovery from deadlock [see Hol 71].

The second contribution concerns the simulation of operating systems as an integral part of their design and construction. The idea, which is devoloped in some detail, was first suggested by Zurcher and Randell [Zur 68] who claimed that it should be possible to design and implement a series of simulations which would become the operating system. The thesis shows the feasibility of this approach and provides some of the techniques and principles required for its realization.

The simulation technique of system implementation offers the advantages of providing feedback on the expected performance of the system, avoiding duplication of effort of a separate simulation, and of having simulations which are necessarily valid for the resulting implementation. The simulation technique is based on the concept of an activity based simulation language and the notion of a resource semaphore mentioned above.

The third contribution of the thesis is in an area which has been widely referred to as design methodology or structuring the complex sequence of steps leading to an implementation. Here we are concerned with the effective and systematic use of the ideas cited above. The notion of a resource and the idea of constructing an operating system from a sequence of simulations must be combined so that they result in a sequence of steps, each small enough to be implemented easily and efficiently.

1.3 Organization of the thesis

The body of the thesis is devoted to presenting the contributions outlined above in more detail. Chapter 2 defines a process and classifies a number of different synchronization problems. The resource semaphore is introduced with its associated data structures and operators. Three examples of the use of resource semaphores in the context of an operating system are described in detail.

Chapter 3 explains the concept of an activity based simulation language and shows how such a language can be used in conjunction with resource semaphores to generate a sequence of simulations which become the operating system. Chapter 4 presents a methodology for operating system construction based on the ideas of Chapters 2 and 3.

Chapter 5 describes the implementation of a small Multiprogramming Operating System (MOS) for a hypothetical machine. The operating system is based on the new ideas presented in the thesis and is implemented and simulated at each of four levels. Chapter 6 contains an evaluation of this research and suggestions for future work.

CHAPTER 2. SYNCHRONIZATION AND RESOURCE ALLOCATION

The purpose of this chapter is to introduce a new set of primitive operations for process synchronization and resource allocation. These primitives are generalizations of existing primitives and provide a means of solving problems of mutual exclusion, dynamic resource sharing, and message and data passing. The mechanisms are flexible, easy to use, and suppress many of the tedious details of process interaction.

One of the most fundamental concepts in the field of operating systems engineering is that of a process. A process represents the basic unit of work in an operating system and is characterized by the fact that it operates independently of other processes in the system except for short periods of explicit intercommunication. This chapter is concerned with different types of process-process interaction and their implementation.

Direct interactions take place for the purpose of coordinating synchronous activities. If two activities, such as those represented by the processes of two separate users, are completely unrelated, then there is no need to synchronize the activities and therefore no direct communication is needed between the two processes. However, if two processes are performing highly related activities, such as when they are both processing the same file or when one is performing a service for the other, then the two activities must be synchronized so that processing can proceed in an orderly fashion.

Resource allocation is a form of indirect process-process interaction. When the demand for a particular resource exceeds the supply, then a decision must be made to curtail certain activities demanding the resource and to let others proceed. When one activity frees the resources it has been using, another activity, which was previously curtailed because of a lack of the resource, may be restarted. Indirect interaction is characterized by processes which are functionally independent, but must cooperate due to finite resources within the computer system.

The remainder of the chapter presents a more detailed definition of the notion of a process and classifies different types of process-process interaction. It then briefly describes existing primitive operations which have been used for process synchronization. A new notion of a generalized resource is introduced and its relationship to process synchronization is shown. We then describe the data structures and primitives necessary to implement this concept and compare them to existing ones in terms of generality, ease of use, and efficiency.

2.1 Definitions

Because the notion of a process (sometimes called a task) is fundamental to operating systems, many attempts have been made to define it. Most of these are informal operational definitions which try to capture the essence of what a process does rather than what a process is. The process concept is dificult to define because processes exist and change in time. Formal definitions of processes try to capture the state of the process at discrete points in time and the way in which a process is allowed to communicate with its environment (other processes in the system).

A <u>process</u> is defined informally as a procedure (set of instructions) in execution by a processor. The <u>state vector</u> consists of the programs and data associated with the process at a particular instant in time. The state vector may also include the states of certain peripheral devices which the process is using, as well as the contents of various processor registers. The description of a process is often compacted into a data structure which contains references to the elements of the state vector. This structure is usually called a <u>process control block</u> (PCB) and includes such things as the contents of the instruction counter and other machine registers, pointers to program and data parts, pointers to input/output control information, process status, and information about resources owned by the process.

SEQUENCING OF PROCESSES

Processes may be thought of as having active and passive phases. A process is <u>running</u> when it controls a processor and is executing instructions on the processor; it is <u>passive</u> otherwise. To maintain the highest possible utilization of processors, when a process changes from running to passive state, another process (if any) must change from the passive to the running state. This change of running process is one of the most important operations in an operating system and is called a <u>process switch</u>. The number of running processes is always less than or equal to the number of main processors. A process switch must save the processor dependent part of the state vector of the currently running process (in its process control block for example) and restore the processor dependent part of the state vector of the next running process. The processor dependent part of the process consists of the registers in the processor which generally include the instruction counter and various general purpose registers.

The state of the currently active process must be saved so that during its next active phase it may be resumed at exactly the point at which it left off (as indicated by the instruction counter) with exactly the same temporary data items (as stored, for example, in any general purpose registers). The primary characteristics of this type of sequencing is that it is symmetric in that there is no hierarchical relationship or dynamic nesting between the current process and the next process (as in procedure calls) and that the entry point of a process is variable and is always the instruction following the last point of exit.

EXECUTION STATES

In general, processes interact with each other in two ways. They interact indirectly by modifying portions of their state vectors which are shared with other processes. They interact directly by sending signals to each other which change the execution state of a process. The execution state of a process is a Boolean variable indicating whether a process is logically ready (the dynamic progress of the process is logically permissible) or logically blocked (the dynamic progress is not permissible). While a process which is logically ready may be running or passive depending on whether a processor is available, a logically blocked process is always passive. Those processes which are logically ready constitute the ready list.

It is clear that at least two operations are needed to implement and study the explicit interaction between processes mentioned above. Namely, we need one to change the status of a process from logically running to logically blocked and one to change the status of a process from logically blocked to logically running. Since only active processes are able to execute these operations, a process must make arrangements to be awakened (have its state changed to logically ready) before it goes to sleep (has its state changed to logically blocked). Section 2.3 reviews some of the existing primitive operations for process interaction while Sections 2.4 to 2.9 describe a new set of primitives which link the notions of process synchronization and resource allocation. To motivate these later sections, however, it is helpful first to explore more deeply the different kinds of process synchronization which take place and the role this interaction plays in operating systems.

2.2 Classification of synchronization problems

It is possible to distinguish three kinds of process interaction:

(1) mutual exclusion

(2) dynamic resource sharing

(3) message and data passing

These three categories are not necessarily mutually exclusive but are believed to include all synchronization problems in operating systems.

MUTUAL EXCLUSION

If two processes are sharing the same data base (i.e. the data base appears as part of the state vector of each process) then each process should insist on exclusive access to a variable before updating it. To demonstrate how parallel access to a data item may cause erroneous results consider the example of a computerized airline reservation system in which the following sequence of events occurs:

(1) Ticket salesman A interrogates the system and finds that 1 seat remains on a flight with a capacity of n people.

(2) Salesman B interrogates the system and also finds the number of reservations is n-1.

(3) Salesman A confirms a reservation for his customer and changes the number of confirmed reservations to n.

(4) Salesman B also confirms a reservation for his customer and changes the number of confirmed reservations to n.

While the system records n reservations, n+1 tickets are outstanding for a flight of capacity n. A similar situation tion occurs when two processes execute the following three instructions in parallel to increment a counter:

```
LOAD    X
ADD     1
STORE   X
```

All that is required here to obtain erroneous information is that one process executes the first instruction while another process has executed statement 1 but not yet executed statement 3. It is not necessary for the parallelism to be real. Even in a uniprocessor environment it is possible that a running process becomes passive (though still logically ready) after the processing of an interrupt. In the example above process A could be interrupted between the first and second instruction and then the scheduler might choose to resume process B which would execute all three instructions. When the last two instructions were finally executed by process A, it is using the obsolete temporary data which it stored in a machine register and the final result is that the counter is incremented by one instead of two.

The problem of coordinating a number of asynchronous activities was studied in detail by Dijkstra in his monograph entitled "Cooperating Sequential Processes" [Dij 65]. The following definitions are derived from this work. Sections of code which access portions of the state vector which intersect with state vectors of other processes are called <u>critical sections</u>. The problem of preventing simultaneous execution of critical sections is called the mutual exclusion problem.

Before entering a critical section a process must invoke a mechanism which decides whether the process is permitted to enter. If we refuse to allow processes to wait by looping, the execution state of processes not permitted to enter their critical sections must be changed from logically running to logically blocked and waiting to enter a critical section. When a process completes the execution of its critical section, another mechanism must be invoked to awaken a process (if any) which is logically blocked and waiting to enter the critical section. Each critical section is associated with a particular data item so that it is possible for two processes to be in critical sections relative to different data items.

It is important to note that these mechanisms must be used by convention between processes to prevent simultaneous access to the data. In other words a process must know that it is competing with another process or processes for access to a data base. Otherwise there is no need to use the mutual synchronization mechanisms. If competing processes fail to observe the conventions regarding mutual access, the results will be unpredictable.

RESOURCE SHARING

Another reason that processes are time dependent is for the mutual sharing of the system's resources. The most common kind of system resources are those that are serially reusable. A serially reusable resource is used by one process at a time and is allocated to the process from a resource pool. When the process has finished using the resource it returns it to the pool and it is then ready for allocation to another process. Examples of serially reusable resources are core or auxiliary storages; peripheral devices such as printers, plotters, punches, and tape units; and certain modules of code which are self-initializing. Obviously a synchronization mechanism is needed for the controlled use of serially reusable resources. When a process requests a resource of a particular kind and that resource is already in use, there must be some mechanism for changing the execution state of the process to logically blocked and waiting for the resource.

As is the case for mutual exclusion, the synchronization between processes competing for resources is indirect. That is, a process should not have to know what other processes are competing with it for a particular resource. The implication of this is that when a process releases a resource or exits from a critical section, the burden of awakening the next process to use the resource or enter its critical section should lie with a system routine or process rather than the implementer of the user process.

Serially reusable resources may be classified as being used exclusively by one process or shared by a number of processes. If there are no restrictions on the sharing of a resource then there are no synchronization problems since any process that requests the resource gets it and proceeds. In this case "resource" is somewhat a misnomer since there is not a limited supply. The concept is sometimes still useful however in keeping track of the processes that are using the resource. For example a data base might be a shared resource to those processes wishing to read it, but used exclusively by a process wishing to modify it.

Resources which are not serially reusable will be called consumable. Consumable resources are those resources which when allocated to a process are not returned to the system. A buffer pool is a good example of a consumable resource if we consider the buffers divided into two distinct classes, full buffers and empty buffers. Then a process might draw empty buffers from one pool and release full buffers to another pool. If there are no empty buffers available the process must invoke a mechanism which changes its execution state to logically blocked and waiting for an empty buffer. Unlike most serially reusable resources the consumable resources are usually software constructs rather than hardware.

MESSAGE AND DATA PASSING

The final kind of process interaction can be classified as message and data passing and is used by one process to send information to another process. Messages are usually passed in buffers reserved especially for this purpose. Associated with a process is a message queue which contains messages sent from other processes. A process may service this queue by receiving the next message and taking the appropriate action. If there are no messages in the queue of the process and the process makes an unconditional request for the next message on its queue, the execution state of the process must be changed to logically blocked and waiting for a message. Messages may be used to delegate tasks to subordinate processes. They may also be used to coordinate the use of resources. An example of message and data passing occurs in a time sharing system with a message interpreter process which cyclically: (1) receives a message containing a command from a user terminal, (2) interprets the command, and (3) sends a message to wake up a process which performs the action associated with the command.

It is hoped that the preceding discussion pointed out some of the problems which exist in operating systems that require synchronization. The following section deals with various existing solutions to the problem of synchronization of processes.

2.3 Existing synchronization primitives

The problems of synchronizing parallel processes came into being when it was realized that a single processor could be used to carry on many computations "simultaneously" by sharing the processor's time among the many computations. Also the notion of having many central processors has intrigued computer scientists for many years in spite of the fact that multiprocessing hardware has come into widespread use only recently.

WAKEUP AND BLOCK

An early attempt at specifying a general approach for coordination of processes was made in connection with the MULTICS project at M.I.T. [Sal 66]. The primitives Saltzer proposed are called wakeup and block. They are presented here with some minor alterations by Lampson [Lam 68].

The wakeup primitive has a parameter p and its function is to change the execution state of the process p to logically ready. The block primitive also has a parameter p and its function is to change the status of the process to logically blocked. When the object of a wakeup is already awake, it remembers the wakeup by means of a wakeup waiting switch (WWS) and ignores the next subsequent block which is directed to it. More formally the primitives are defined as follows:

> Wakeup(p): If P is logically blocked change its execution state to logically ready. Otherwise set its WWS to true.

Block(p): If WWS of P is <u>true</u> then set WWS to <u>false</u>. Otherwise change the execution state of P to logically blocked.

The flaw that any given process can remember only one wakeup signal was reported by Rappaport [Rap 68]. Although a one wakeup memory is satisfactory for most applications, it is convenient to replace the wakeup waiting switch by a counter indicating at any given time the number of wakeups the process has received less the number of blocks since the last period of inactivity.

These primitives are the most rudimentary in the hierarchy of operations to be presented in this section because the processes executing them must have complete knowledge of its environment. A process executing a wakeup or block must know explicitly to which process the message is being directed. When processes are cooperating in a dynamic system, i.e. one in which processes are being created and destroyed, each must have access to the data structures which describe processes currently in the system.

It is also necessary for the processes to keep track of the conditions or state variables which enable a sleeping process to be awakened or whereby a ready process should be put to sleep. These state variables must be shared between the process setting the condition for reawakening and the process detecting that the condition has been satisfied. The integrity of this data must therefore be protected by some kind of mutual exclusion primitives. In MULTICS these primitives are called <u>lock</u> and <u>unlock</u> [Sal 66].

Thus wakeup and block are rudimentary in two major areas. First they require explicit communication between processes which makes its direct use difficult in the application areas of mutual exclusion and resource allocation. Second they require that all the bookkeeping associated with putting a process to sleep and reawakening that process are done separately from the action of the primitive. Wakeup and block are, however, useful in the construction of more powerful primitives for process-process interaction.

DIJKSTRA'S PRIMITIVES

Dijkstra's primitives for process synchronization [Dij 65] are more sophisticated and allow each process to act more independently of its environment. Namely, the state variables (semaphores) which are the operands of the primitives contain information about the environment. The primitives Dijkstra uses corresponding to block(p) and wakeup(p) are P(sem) and V(sem), where sem is a semaphore. A <u>semaphore</u> consists of an integer variable and a list of processes. The integer variable is initialized by the system and the list of processes is initially empty. Subsequent access to the semaphores is only through the two synchronization primitives described as:

P(sem): Decrement sem by 1. If sem < 0 book the invoking process on the waiting list for sem and set its status to logically blocked.

V(sem): Increment sem by 1. If sem <= 0 remove one of the processes on the waiting list for sem and change its status to logically ready.

To insure the integrity of the data associated with the semaphore, the semaphore must be accessed by only one process at a time. It is therefore necessary to make the P and V primitives indivisible, i.e. if two operations should be invoked simultaneously on the same semaphore, the effect will be that they are executed sequentially in an undefined order.

These primitives are more sophisticated in the sense that they have added some structure to the society of processes by partitioning the set of logically blocked processes into equivalence classes each of which consists of all the processes waiting for a single semaphore. This semaphore is the object of the suspension imposed on the waiting process. Furthermore, these processes may be reactivated only by executing a V operation on the associated semaphore.

Whereas the lower level forms of the synchronization primitives have no data structures set up to keep track of and reactivate logically blocked processes, the implementation of the P and V primitives not only structures the blocked processes, but also introduces an algorithm as part of its implementation for scheduling processes waiting for a semaphore when a V operation is executed. This mechanism tends to insulate the processes from their environment in that individual processes can be written independently of any other process. The execution of a P operation implies potential blocking or delay for the invoking process (and only the invoking process) while execution of a V operation implies the removal of a barrier for a process which need not be specified by the invoking process.

IBM'S ENQ, DEQ, WAIT, AND POST

IBM provides four primitives for process synchronization [IBM 68a] in the form of macroinstructions. ENQ and DEQ (read enqueue and dequeue) are similar in many respects to P and V, but are more restrictive in one sense and more general in another. The parameter of an ENQ or DEQ is the address of an 8 character alphanumeric string which serves to identify the object of the operation and is called a resource. Associated with each resource is a list of processes which have ENQed on the resource and represent both processes using the resource and processes logically blocked and waiting to use the resource. ENQ is more general than P because the invoking process can specify whether it wants shared or exclusive use of the resource and whether the request is conditional or unconditional. If the request is grantable, the process is queued on the list associated with the resource and remains in the logically ready execution state. If the request is unconditional and not grantable, then the process is queued on the list and its execution state is changed to logically blocked. If the request is not grantable and conditional, then the process is alerted that the resource was not available.

ENQ and DEQ are more restrictive than P and V because they must be strictly paired so that within a process every ENQ on a resource must be followed in a computation by a DEQ and no DEQ can be executed before an ENQ on the resource. This imposes the rather severe restriction that each "resource" can represent only a single resource item such as one tape unit, one block of storage, or permission to access one data item or data base. The counter associated with the resource semaphores of P and V, however, can conveniently be used to count the number of resources in a pool of resources. If the counter is positive then it represents the number of resources available in the pool. If negative it represents the number of processes waiting for the resource.

The WAIT and POST macroinstructions are analogous to block and wakeup. WAIT operates on an event control block (ECB) and places the process in a logically blocked execution state unless the ECB has previously been POSTed. POST also operates on an ECB and awakens a process if one is waiting on the ECB. If no process is waiting then a flag is set in the ECB indicating that the event was POSTed. An event control block can only be reused after the flag has been reset indicating that the event has not yet occurred. A generalization of the WAIT macroinstruction allows a process to wait for the POSTing of n of m possible events where n is less than or equal to m. The primary drawback of the WAIT/POST set of primitives is that it is impossible for more than one process to wait for a single event. This means that WAIT and POST are restricted in their use primarily to cooperation in which one process signals the completion of some activity to a second process.

BRINCH HANSEN'S PRIMITIVES

Some sets of synchronization primitives have been oriented to the third category of synchronization problems, namely message and data passing. The philosophy here is that an interprocess message handling facility will serve as the basis for solving the synchronization problems in the other two areas. We will use as an example of the message oriented primitives those used by Brinch Hansen [Bri 69] which were developed in connection with the RC 4000 multiprogramming system of A/S Regnecentralen, Copenhagen, Denmark.

Brinch Hansen's four primitives have the form:

send message(receiver,message,buffer)
wait message(sender,message,buffer)
send answer(result,answer,buffer)
wait answer(result,answer,buffer)

Send message gets a fixed length message buffer from a buffer pool, places the specified message into the buffer, and then queues the buffer in the message queue of the specified receiver of the message. The sending process is informed of the location of the message buffer so that it can wait for an answer in the same buffer. In addition to the message, the message buffer contains the identity of the sender. If the intended receiver of the message is no longer in the system, the sender is sent a dummy answer by the system. If the receiver of a message is waiting for a message, its execution state is changed from logically blocked to logically ready.

The wait message primitive suspends the calling process until a message arrives in its queue. When a message arrives, the process is restarted and the name of the sender and the contents of the message are copied into the data area of the receiving process. The message buffer is removed from the queue and its location is returned to the calling process so it can be used to transmit an answer. If the sender no longer exists in the system the caller is so notified.

The send answer primitive copies an answer into the same buffer in which a message was received and places it in the queue of the original sender. The sender's execution state is changed to logically ready if it was waiting for an answer. The answering process can also specify whether the answer is a normal or a dummy answer. Dummy answers are sent when messages are rejected or unintelligible or because of receiver malfunction.

The wait answer primitive is analagous to the wait message primitive except that on receipt of the answer, the buffer is returned to the buffer pool.

It is clear that Brinch Hansen's primitives impose very rigid conventions for process communication. Conversations may only take place between two processes, the receiver of messages must be explicitly named, every conversation consists of four distinct operations, and every message must be acknowledged. This strict regimentation of conversations may make the interactions between processes easier to analyze and understand. This may in turn make operating systems easier to build and debug. On the negative side, these primitives are inconvenient for handling relatively easy synchronization tasks such as mutual exclusion and resource allocation. Consequently they would probably be used in conjunction with other synchronization primitives for these purposes.

OTHER PRIMITIVES

Because every multiprogramming system requires a mechanism for process synchronization, and because there have been no standards or paradigms in this area, there are almost as many synchronization primitives as there are operating systems. Most are essentially similar to one of the four types mentioned above, differing only in the details of their implementation. For example the latest interprocess communication facility of MULTICS [Spi 69] is very similar to Brinch Hansen's scheme. In MULTICS, communication of processes can only be achieved by an exchange of messages in a commonly accessible "mailbox" whose identity is known to the user processes by convention. A <u>notify</u> primitive returns the first message to be put in the mailbox, if any, and blocks the process otherwise. The primitives called REQUEST and RELEASE in the ESOPE multiaccess system [Bet 69] are identical to Dijkstra's P and V.

In the process control and communication facility of the GE 600 operating system [Ber 69] the shared objects are called <u>events</u>. A process can request notification when an event occurs by issuing a NOTIFY primitive. The event is said to occur when some process issues a CAUSE primitive to it. The CAUSE primitive activates <u>all</u> processes waiting for a particular event. (This also happens implicitly when a process DEQs on a resource which it held exclusively and which several processes are waiting to use in shared mode.) The GE facility also allows the user to specify attributes of an event such as the maximum waiting time or the maximum queue length. This concept of events is basically a generalization of WAIT and POST.

2.4 Resource allocation and logical resources

The foregoing discussion was meant to provide the motivation and background for an essentially new approach to the problems of synchronization and resource allocation. The approach requires that we modify somewhat the traditional notion of a resource so that the connection between synchronization and resource allocation becomes apparent.

Almost all resource allocation systems require that if the demands of the requesting process cannot be satisfied the process is put on a queue and put to sleep until the resource becomes available. This would usually be done by the invocation of one of the blocking primitives mentioned above - block, P, ENQ, WAIT, wait message, or wait answer. P and ENQ have the queuing mechanism built in so it would seem that they are tailor made for this form of synchronization. But it is not sufficient for these primitives to simply queue the waiting processes. What is needed is a flexible allocation policy so that when a process returns a resource, possibly removing a barrier to a waiting process (by a wakeup, V, DEQ, POST, send message, or send answer), it is possible to choose a process from among those waiting for the resource. The V and DEQ mechanisms do not provide this flexibility. The schedulers associated with the operations are fixed when the primitives are implemented. This means that to use these primitives alone for resource allocation would restrict the system to one allocator for all types of resources.

One alternative which can be used is to create a process whose sole purpose is to allocate resources to requesting processes. This process then receives signals (wakeup, POST) or messages (send message) from those processes requesting resources and having made the allocation returns a signal or message to the requesting process. After sending a requesting message or signal a process must wait for a signal or answer before it can use the resource that is requested. A process would have to send another signal or message when it wished to return the resource to the system. Since P and V can be used for sending signals between processes they can be used in this respect also.

GENERALIZATION OF THE NOTION OF A RESOURCE

Whenever a process goes to sleep or is put to sleep it is presumed that it went to sleep because it logically could not proceed until an event external to the process has occurred. In the preceding sections we always used the terminology "the execution state of the process is changed to logically blocked and waiting for X" where X is the object of the blocking. This choice of words was used repetitiously to motivate the following definition of a resource.

> A <u>logical</u> <u>resource</u> is anything which may cause a process to become logically blocked.

Thus logical resources would include traditional resources such as tapes, disks, drums, core storage, and channels. It would also include such things as empty and full buffers, the completion of an I/O operation, the passage of n units of real time, the event of being 12 noon, a signal or message from another process or processor, or the pressing of a button on the operator's console. A process may block itself to wait for any of these logical resources.

The traditional resources are referred to as hardware resources. Those logical resources which are not hardware resources are called software resources. The reason for introducing the logical resource concept is to enable us to deal uniformly with all such resources and so that we may introduce mechanisms or primitives which permits both the synchronization of cooperating sequential processes and the "allocation" of logical resources.

By introducing the notion of a logical resource we are able to say that the only way a process may become logically blocked is by requesting such a resource and hence that all logically blocked processes are waiting for a logical resource. This in turn structures the society of cooperating sequential processes in a uniform way and permits easier analysis and understanding of such systems. The logical resource concept also provides a framework for solving problems of optimal resource allocation strategies. These problems can be approached on a global level for solution by analytical means or through simulation.

2.5 Data structures for resource semaphores

The key to the success of the primitives to be presented in the following sections is the specification of data structures which consist, in part, of executable code. This technique allows the flexible handling of any number of different structures of the same basic type.

Queue structures are an excellent example. The two basic operations that one wants to perform to alter a list are insertion of an element and removal of an element. The exact nature of these operations, of course, depends on the queuing discipline, whether last in first out (LIFO), first in first out (FIFO), simple priority ranking, multiple priority ranking, whether the list is singly or doubly linked, and how the end of the list is flagged. Also it may be necessary or desirable to collect statistical information regarding the usage of the queue. This might include the average and maximum length of the queue, or the average and maximum time in the queue for any particular element. All these variables must be taken into consideration when code is executed to insert an element into the queue or remove an element from the queue.

One would like to take advantage of many different kinds of lists in a large software system while maintaining the convenience of writing tho same calling sequence in each case. Often the variability is handled through a macro facility so that the queue discipline is specified as a parameter in the macro call. The macro expansion then consists of the proper code for inserting or removing an element from a queue of a particular discipline.

There are two difficulties with this approach. First, to include all the degrees of variability mentioned above takes a great deal of parameterization of the macro, making the call difficult to write and understand. Without all the parameterization one is restricted to a small number of modes of operation. Second, the mode of operation is bound when the macro call is written. To change the statistical gathering capabilities (for debugging for example) it would be necessary to change all the macro calls.

By including executable code as part of the queue data structure it is possible to use the same calling sequence for all queues. Fig. 2.1 shows the data structure required for a queue constructed in this way. It consists of a pointer (absolute address or segment number in a segmented name space) to the code to insert an element, a pointer to the code to remove an element, and a list header which would minimally contain a pointer to the first element on the queue. The header could be augmented as shown by the last element on the queue and other indicators and statistics associated with the queue. The calling sequences for all queues might be, say

```
INSERT(Q,E)
REMOVE(Q,E)
```

to insert and remove an element E from the queue Q. The result of invoking these primitives would be to call the

```
QUEUE
  |
  |        ┌──────────────┐
  └---->|       •------+-----> INSERT ROUTINE
        ├──────────────┤
        |       •------+-----> REMOVE ROUTINE
        ├──────────────┤
        |       •------+-----> FIRST ELEMENT
        ├──────────────┤
        ├──────────────┤
        |       •------+-----> LAST ELEMENT
        ├──────────────┤
        |              |
        | INDICATORS   |
        |              |
        |     AND      |
        |              |
        | STATISTICS   |
        |              |
        └──────────────┘
```

Figure 2.1 Queue data structure. Items below the double horizontal line are optional.

```
RESOURCE SEMAPHORE
  |
  |        ┌──────────────┐
  └---->|       •------+-----> AVAILABLE RESOURCE LIST
        ├──────────────┤
        |       •------+-----> WAITING PROCESS LIST
        ├──────────────┤
        |       •------+-----> ALLOCATOR
        └──────────────┘
```

Figure 2.2 Basic resource semaphore data structure associated routines with parameters Q and E.

The element E is a parameter in the REMOVE primitive because the queue discipline does not always determine in what order elements leave the queue. For example a list of core storage elements might be ordered according to size, but removal from the list would be based on the size of the block requested. Thus it is not always the first or last element that is removed from the list, so the choice of the element to be removed is made by the calling routine.

Because the code associated with insertion and removal is accessed indirectly through the queue data structure, this mode of operation is slightly more inefficient than in line code or a single subroutine call. However it is believed that the power and flexibility achieved by having data structures with executable parts more than compensates for this slight penalty. At worst the penalty paid is an extra subroutine linkage for each call to the primitives. At best, if there is an instruction permitting indirect branching the cost will be as low as one machine cycle.

RESOURCE SEMAPHORE DATA STRUCTURES

The data structure used to implement a logical resource is called a <u>resource semaphore</u>. The basic components of a resource semaphore are shown in Fig. 2.2. It consists of three entities:

(1) available resource list

(2) waiting process list

(3) allocation routine

The available resource list consists of the *logical resource elements* currently in the inventory of the *logical resource class* represented by the resource semaphore. The data structure for a logical resource element is called a *resource control block* (RCB). The waiting process list consists of those processes logically blocked and waiting for the logical resource.

The allocator is a routine which has as its implicit parameters the resource list and the waiting process list. The allocator decides which if any of the resources should be allocated to the waiting processes. For each call to the allocator either no allocations are made or one or more resources is allocated to one of the waiting processes. The resources which are allocated and the process to which they are allocated are removed from their respective lists and the address of the PCB is the value of the allocator function.

Executable data bases permit the structure of the two lists to be left completely open until the resource semaphore is created. This means that it is possible to accommodate a large variety of list structures for resources. This is particularly convenient because some resource classes have indistinguishable resources (empty buffer pools, core blocks of equal size) while others have distinguishable resources (full buffer pools, core blocks of varying size). These resource classes require different handling.

Either or both lists of the resource semaphore may be empty, but more importantly both lists may be non-empty at the same time. This may occur for example if the available supply of resources is not sufficient to meet the demand of any one process or if the allocator routine decides that, although possible, an allocation is not desirable. The decision to withhold available resources may be based on knowledge that the allocation would cause the system to be in danger of deadlock or that a request of higher priority is imminent.

2.6 Primitive operations on resource semaphores

The two operations on resource semaphores will have the form:

REQUEST(RS,DATA)

RELEASE(RS,RCB)

where RS stands for Resource Semaphore. The operations are analogous to P and V respectively since REQUEST implies potential blocking and RELEASE represents the possible removal of a barrier for another process. REQUEST is a function that returns a pointer to the RCB of a resource of the type requested. The DATA parameter specifies information about the request if the request is qualified in any way, and is optional. The basic philosophy in specifying these primitives has been to extract those functions common to all process interaction and leave variable (as part of the allocator) those functions unique to particular applications.

In its simplest form the algorithm for the REQUEST primitive is shown in flowchart form in Fig. 2.3. First the request data is stored in the PCB of the process. Since this information must be retained while the process is blocked and waiting for the logical resource it must be stored as part of the PCB. Next the state of the process is changed from logically ready to logically blocked. The process must be logically ready to be executing the request and this fact is indicated by its membership in the ready list. Its logically blocked status is indicated by its membership in the waiting

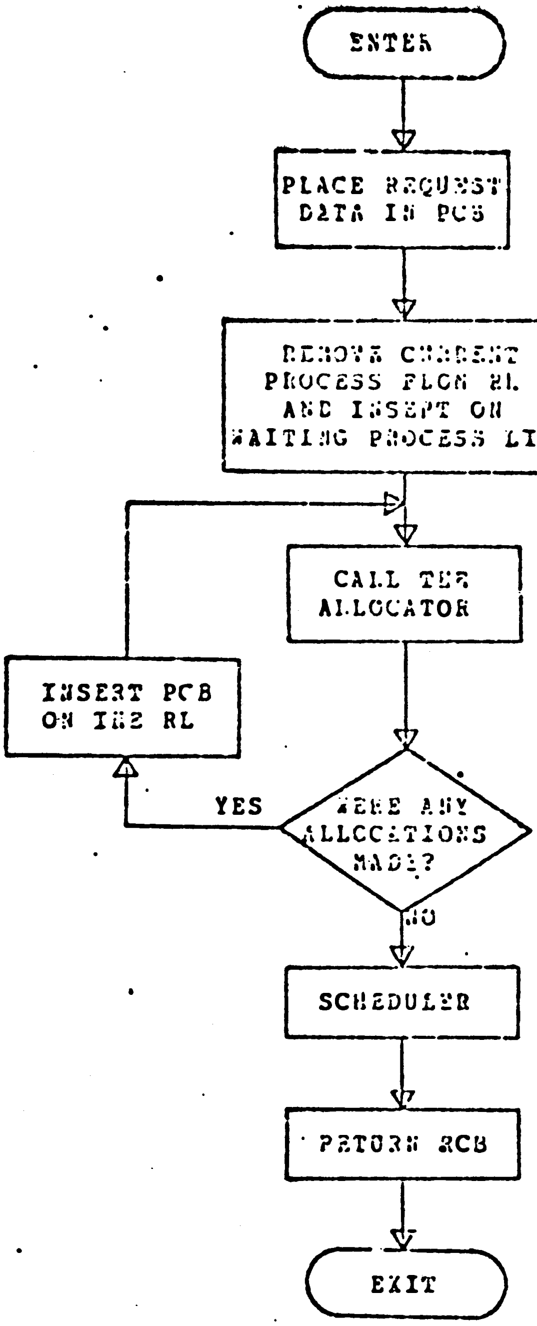


Figure 2.3 Flowchart of REQUEST

list of a resource semaphore. We adopt the convention that a process can be on only one list at a time, namely the ready list or waiting list associated with some logical resource. This means that the list can be threaded through the PCB of the process without the use of extra storage.

The next box in the flowchart indicates the calling of the allocator associated with the resource semaphore. The allocator has access to both the resource list and the waiting process list (and possibly other relevant system information) and based on the rules of allocating the resource makes an allocation by storing the address of the allocated RCB in the PCB of the process to which the allocation is being made. The process and the allocated resource are removed from their associated lists. The next step in the algorithm asks whether any allocation was made in the last iteration (as indicated by the return parameter or the allocator). If an allocation was made, the process to which the allocation was made has its status changed to logically ready by its insertion on the ready list. The allocator is then called again to see if another allocation is possible. This loop continues until no allocation is possible. When the loop is exited in this fashion, the process scheduler is called. The function of the process scheduler is to choose one of the ready processes to run on the processor. In fact, the process scheduler is the resource allocator for the CPU.

If the scheduler chooses a process that is not the current process (the one which executed the request in the first place) then the scheduler performs a process switch to the chosen process which then becomes the running process and controls the CPU until it is interrupted or until it executes a REQUEST or RELEASE. There are two ways that it is possible for the process executing the REQUEST not to be the process chosen by the process scheduler. First the process may not be allocated the resource it requested so that it remains in the logically blocked state on the list of waiting processes. Second, if it is allocated a resource and is returned to the ready list it may be returned to a different position on the ready list and hence not be chosen for execution. This would be the case if the ready list were FIFO. Moreover if more than one process has its execution state changed from logically blocked to logically ready, there are more processes for the scheduler to choose from.

Assuming that the scheduler does choose a process other than the current process, the next box in the flowchart is not immediately executed. Instead, the stateword of the currently executing process is saved in the current PCB (the instruction pointer points to the next flowchart box) and the stateword of the process chosen for execution next is retrieved from its PCB and it is then resumed where it left off previously. If the current process is still on the ready list, then the scheduler on some subsequent invocation resumes the process for its next active phase. If the current process is on the blocked list waiting for a resource then it remains there until, during the REQUEST or RELEASE invoked by another process it is removed from the waiting list and placed on the ready list. In either case the process is resumed at the last flowchart box in Fig. 2.3.

When the process is finally resumed, the pointer to the allocated RCB is retrieved from its PCB and is returned as the value of the REQUEST operation. The allocation pointer must be stored in the PCB by the allocator so that it is remembered during the interval of time that the process is logically ready, but passive.

The structure of the RELEASE algorithm is very similar to that of the REQUEST algorithm and its flowchart is given in Fig. 2.4 First the resource specified by the second parameter is placed on the resource list of the resource semaphore. Next we execute the same loop as in the REQUEST primitive with each allocation resulting with another process having its execution state changed from logically blocked to logically ready. The execution state of the process invoking the RELEASE is not changed in any way. When the scheduler is called we again have two possibilities. Either no process is made ready and the currently executing process remains in control of the CPU or the scheduler may choose one of the other ready processes and a process switch takes place. Again a process switch implies a saving of the stateword of the executing process and a restoring of the stateword of the process being switched to. If the currently executing process becomes passive, then when it is again activated by a subsequent call of the scheduler, the algorithm is resumed at

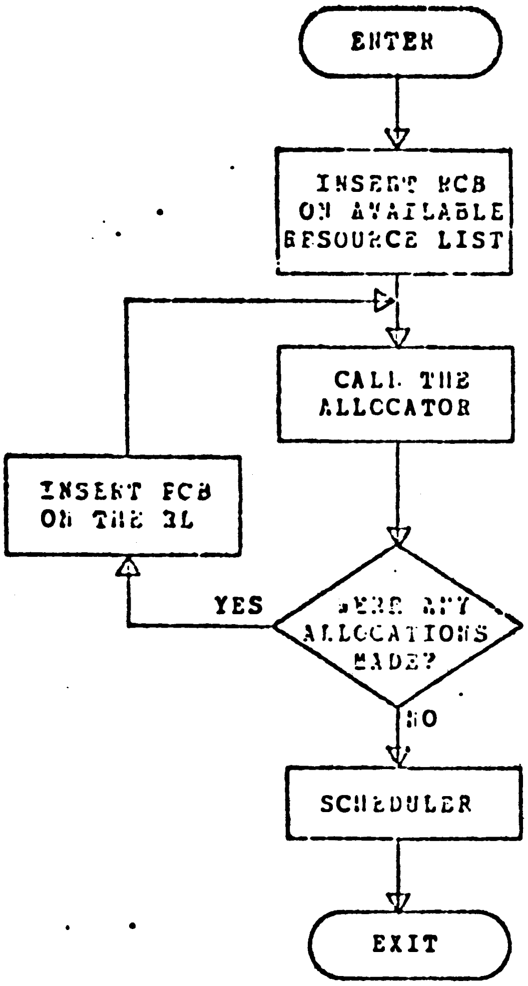


Figure 2.9 Flowchart of RELEASE

the flowchart box following the scheduler in this case a return from the RELEASE primitive.

The scheduler box of the RELEASE flowchart is not crucial to the algorithm. If it were eliminated, this would mean the process executing a RELEASE would retain control of the processor and would thus remain active. This implementation would be slightly more efficient, but would deny the flexibility of being able to switch to a higher priority process in the event that one were made ready by the release of a resource. The same considerations applies in the implementation of Dijkstra's V operation or in the implementation of external interrupt handlers. In the case of an external interrupt such as an I/O interrupt, the process waiting for the I/O operation has its execution state changed to logically ready. Thus the choice is to either automatically return control to tho interrupted process or call a scheduling mechanism to decide which ready process should be executed next.

ALLOCATION STRATEGIES

While specific examples of resource semaphores are given in Section 2.8, it is instructive here to point out exactly what is gained by having a variable allocator as part of a synchronization-resource allocation primitive operation. First such a feature enables the handling of a wide variety of logical resources. We distinguish three allocation strategies:

(1) basic allocation strategy

(2) match allocation strategy

(3) general allocation strategy

In the <u>basic allocation strategy</u> the resources may be null resources and REQUEST and RELEASE are functionally equivalent to P and V. The semaphore value is equivalent to the number of elements on the resource list if positive and the number of processes on the waiting list if negative. The allocator then makes an allocation if and only if there is a null resource element on one list and a process element on the other list. Thus REQUEST and RELEASE can be used to handle any synchronization problem handled by Dijkstra's primitives.

For the case of simple logical resources in which there is no qualitative difference between elements, the simplest strategy for the allocator would be to make an allocation if and only if there is one or more elements on each list. When there are no processes executing either a REQUEST or RELEASE on a resource semaphore which uses this strategy, then either one or both of the resource and waiting lists will be empty. This happens because the allocator continues to assign resources to processes until either there are no more processes or there are no more resources. The basic allocator is very fast and simple because it need not search either list. To make an allocation the first element of each list is removed.

Another possible allocation strategy is called the <u>match allocation strategy</u> in which a particular resource satisfies the request of one particular process or set of processes. The logical resources are qualitatively different so that it is necessary for the allocator to search through both lists to see whether an allocation is possible. For resource semaphores using the match allocator the requesting processes must specify the nature of their request by using the DATA parameter of the REQUEST primitive. The match allocation strategy can be used for passing messages to particular processes or for processes waiting for the completion of an I/O operation. A more detailed explanation of these applications is given in Section 2.8.

Finally we have a general allocation strategy. In this case we can handle resources which do not exactly match the requests of the processes so that some transformation must be applied to resources or the request. Consider a storage resource in which processes request a contiguous block of storage of a particular size. Suppose that the resources are maintained in a list of blocks. The allocator may use a first fit or best fit strategy. If used in an environment in which programs and their data are relocatable it may compact the storage that is in use (garbage collect) to make larger contiguous free blocks available for use by requesting processes.

The next several subsections give examples of the resource semaphore in solving a wide variety of synchronization problems including:

(1) conditional and variable requests

(2) ageing of requests

(3) multiple resource classes

(4) Boolean combinations of events

Brief examples are used to illustrate each of the applications.

CONDITIONAL AND VARIABLE REQUESTS

A request is conditional if the process is made logically ready even if the request is not granted. The process making the request would indicate its conditional nature by supplying this information in the DATA parameter. The allocator, upon seeing that the request of a process is conditional would make a dummy allocation if no real allocation were possible. This value would be the value of the REQUEST primitive and would indicate to the invoking process that no allocation for that particular resource was made.

Furthermore it is possible for an allocator to accommodate variable requests, or conditional variable requests. For example a process might request 10,000 to 15,000 words of core storage and the allocator could be written so as to give the process as much core as is available up to 15,000 words if there is at least 10,000 words, otherwise the process remains on the blocked list. If the request were variable and conditional the process would either be allocated the requested resource or would be assigned a dummy resource indicating no allocation was possible. IBM's GETMAIN macroinstruction [IBM 68a] allows conditional and variable requests for core storage by coding the appropriate parameters at compile time.

AGEING OF REQUESTS

The allocator can also "age" requests for logical resources. That is, a strategy might increase the priority of any process that has been waiting longer than a given amount of time so that its chances of being allocated a resource are increased. Ageing is a means of preventing what Holt calls _effective deadlock_ [Hol 71] which occurs when the utilization of a resource is so high that scheduling policies overlook low priority processes in making resource assignments. Unless these low priority processes eventually have their priorities increased they may never be granted a resource and become effectively deadlocked. Ageing has been implemented as part of the priority job scheduler in the CLASP operating system which is a Cornell University variant of IBM's ASP [IBM 68b].

MULTIPLE RESOURCE CLASSES

Consider the problem of handling requests for multiple classes of resources. For example suppose a process wants 2 tapes, a disk, and 256k of core storage. If a resource semaphore is created for each of these resources there may be a problem of unutilized resources. Suppose the process is allocated two tapes and a disk and is then blocked waiting for core storage because there is only 200k available. Unfortunately this process may wait indefinitely controlling the two tapes and a disk because so large a block of storage rarely becomes available. Other processes which could normally run may become blocked waiting for a tape or a disk. One possible solution to this problem is to do all the allocations simultaneously. That is, the process becomes blocked unless the tapes, disks, and storage requested are all available at once. This solution assures a better utilization of resources, but prejudices the system against processes which make large demands on the system because they are less likely to find all the resources available simultaneously.

Resource semaphores can be used to handle requests for multiple classes of resources if we create one resource data structure to represent several given classes of resources. Then the resource list must necessarily contain the RCBs for each class of resources. The requestor must specify the number and type of each resource requested as part of the data parameter of the REQUEST primitive. The allocator decides on the basis of all the pending requests and all the available resources which if any processes should be allocated their request. The types of resources included in such a logical resource must be known _a priori_ and the format of the request data must be fixed by convention between the allocator and the requestor. Some resource classes such as those needed by a user job step are easily combined into one logical resource.

BOOLEAN COMBINATIONS OF EVENTS

Suppose there exists a finite universe, U, of events each of which has the state <u>happened</u> or <u>not</u>-<u>happened</u>. Consider a resource semaphore whose resource list contains the elements of U. The RELEASE operation can then be used to change the state of one of the events and the REQUEST can be used by processes which cannot proceed until a certain combination of these events has happened. The enter routine for the resource list would not add elements to the list, but would merely change the state of the appropriate element. The DATA parameter of the REQUEST primitive would specify the combination of events that need to happen before the process could continue. For example a process could

REQUEST(U,A <u>and</u> B <u>or</u> C)

where A, B, and C are events in the resource semaphore U. Whenever the state of an event is changed by a RELEASE, the allocator would check whether the change would permit any of the blocked processes to be resumed.

For example the events could be the completion of other tasks. If process D could not continue until tasks A and B are completed or C is completed (where A, B, and C belong to a group of tasks represented by the resource semaphore T), it would execute a REQUEST(T,A <u>and</u> B <u>or</u> C) while each of the tasks would execute a RELEASE(T,XH) where XH is an BCB which indicates that event X has happened. A similar approach applies for Boolean combinations of hardware resources. For example, a process could request a disk or two tapes. However this would require that resource semaphores represent multiple classes of resources and might be more cumbersome than handling each individual resource class separately.

CREATION AND DELETION OF RESOURCE SEMAPHORES

It should be possible to create and destroy resource semaphores dynamically. To do this we use the two primitives:

CREATERS(RQCR,QI1,QR1,WPQCR,QI2,QR2,ALLOCATOR)

DESTROYRS(RS)

The CREATERS primitive has seven parameters. The first three parameters specify the resource queue creation routine (RQCR) and insertion and removal routines for the resource queue (QI1 and QR1); the second set of three parameters specify the waiting process queue creation routine (WPQCR) and queue handlers for the waiting process queue (QI2 and QR2); and the last parameter specifies the allocation routine for the resource semaphore. The DESTROYRS primitive has one parameter which references the resource semaphore to be destroyed.

The CREATERS primitive first allocates space for the resource semaphore data structure. It then calls the resource queue creation routine (RQCR) which allocates space for the resource queue, establishes links to the queue handlers (QI1 and QR1), and initializes statistical information. The location of this queue is stored as part of the resource semaphore data base. Next the waiting process queue is created and initialized in a similar fashion. Finally the allocator address is stored as part of the RS data base. Both lists are originally empty so that any resources must be added by subsequent calls to the RELEASE primitive.

The DESTROYRS primitive assumes that both lists are empty and releases the storage associated with these lists and with the resource semaphore. Since both the creation and destruction of semaphores requires storage management and therefore use both the REQUEST and RELEASE primitives, the creation of the storage management resource has to be done "manually" without using the CREATERS primitive. This is a common bootstrap procedure and should not present any conceptual difficulty. It is anticipated that the resource semaphores used by the operating system will be created when the system is generated and exist for the lifetime of the system. The resource semaphores used by user processes will be dynamically created at run time. Also, facilities should be available to modify allocators and queue handlers during execution to accommodate a changing system environment.

2.7 Implementation considerations

REQUEST and RELEASE would be implemented in the straightforward manner indicated in Section 2.6 were it not for the problem of mutual access to the resource semaphore data base and the ready list. The problem occurs during the parallel (or interleaved) insertion or removal of elements from lists. Fig. 2.5 shows what may happen when two processes attempt simultaneous insertion into a list.

A simple solution to the problem is to only allow sequential access to the resource semaphore data base. Conceptually this may be thought of as making REQUEST and RELEASE extended machine operations. The fundamental characteristics of machine operations which make this possible are (1) that they are indivisible so that no interruption of the machine operation may take place until it is completed and (2) if two processors are simultaneously executing instructions which reference the same memory registers, then the instructions are executed sequentially in some hardware dependent order.

The implementation of these two facets of mutual exclusion can be done in a number of ways. The uniprocessor case is dealt with first because it is the simplest. In a uniprocessor system the danger of true simultaneous access is non-existent because only one sequence of instructions can be executed at a time. The problem lies in the interleaved access in which one process may be accessing the data base and

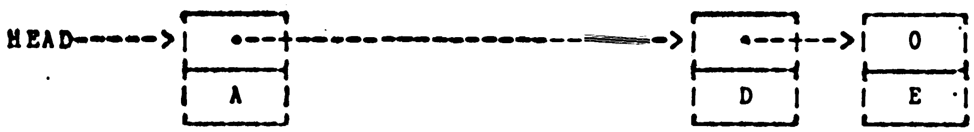


__Figure 2.5(a)__ Original 3 element list

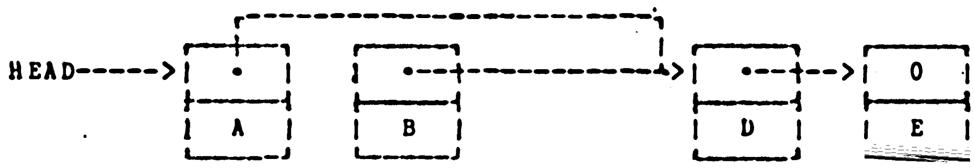


__Figure 2.5(b)__ Process 1 finds new element B is to follow old element A and makes B's successor equal to A's successor.

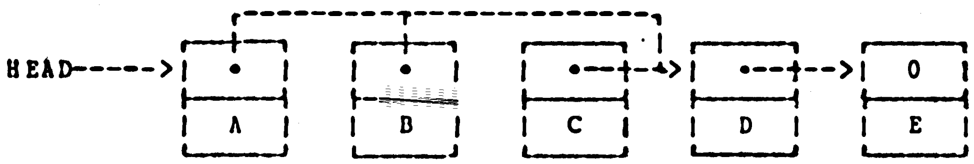


__Figure 2.5(c)__ Process 2 finds new element C is to follow old element A and makes C's successor equal to A's successor.

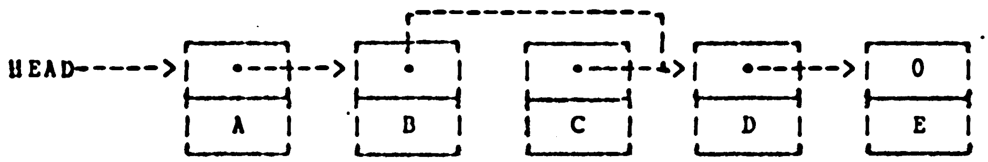


__Figure 2.5(d)__ Process 1 completes insertion of B by making B the successor of A.

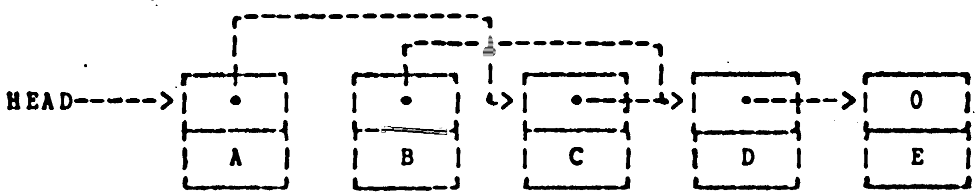


__Figure 2.5(e)__ Process 2 completes insertion of C by making C the successor of A. The final list contains elements A, C, D, and E instead of A, B, C, D, and E.

before the accessing is completed there is a process switch to another process which accesses the same data base. The way to avoid interleaved access in the uniprocessor case is to maintain absolute control of the processor for the duration of the access. Since in most hardware the only way that a process loses control of a processor is by an interrupt, it is sufficient for the process to disable interrupts for the duration of its access.

In the case of REQUEST and RELEASE, if the first instruction executed disables interrupts and the last instruction enables them again, then the entire operation is executed as if one large instruction. Since no other process can execute instructions on the CPU during this time it follows that no other process can execute a REQUEST or RELEASE operation or access the resource semaphore data base.

MULTIPROCESSING CONSIDERATIONS

In a computing system with two or more processors, the solution of inhibiting interrupts is no longer adequate because although a process may control its own processor by inhibiting interrupts, it cannot prevent another process from executing instructions on another processor. The process must somehow inform other processes that it is currently accessing a critical data base. This may be done by setting a lock cell associated with the data base and which is a Boolean variable that can have the values true or false.

Clearly when entering a critical section the lock must be tested and set in one indivisible operation so that it is impossible for two processes to test the lock simultaneously, find it _false_ and then both set the lock to _true_, and both enter their critical sections. Modern hardware such as the IBM System/360 [IBM 68c] provides a test and set instruction specifically for this purpose. The question then arises of what a process is to do if it finds that the lock has already been set by another process. Either the process must loop until the other process completes its use of the critical section and resets the lock (a _busy wait_), or it can go to sleep and arrange to be awakened when the other process resets the lock (an _idle wait_).

To insure that a process does not have to wait for more than a short period of time we must satisfy three conditions:

(1) critical sections are short

(2) critical sections are not unduly delayed

(3) competition to enter critical sections is infrequent

If two processes attempt to enter their critical sections simultaneously, the process that fails must wait at least as long as it takes to execute the instructions in the critical section of the other process, so the critical sections themselves must be short to insure a short waiting time. To insure that the critical section is not unduly delayed (e.g. by pre-emption by a higher priority process) it is necessary to inhibit interrupts during the execution of the critical section as in the uniprocessor case. Finally we want to exclude the case of many processes on different processors all trying to enter a critical section at the same time. This is done by requiring that the probability that a process is waiting to enter its critical section is negligible.

Figure 2.6 is a flowchart for an inhibited busy wait implementation of a lock and unlock which satisfy the three conditions for REQUEST and RELEASE when used with sufficiently short allocators. The trouble is that the busy wait is not fail safe. In other words there is no guarantee that once a process has done a lock on a lock cell that soon thereafter it will do an unlock on the same cell. This would be extremely dangerous since the process executing a lock would have complete control of the processor. This type of mutual exclusion would thus have to be restricted in its use to systems programs which are debugged. As a consequence, since the allocator of REQUEST and RELEASE would be operating in protected mode with respect to the resource semaphore being allocated (and hence with interrupts disabled) it would be inadvisable to let user processes specify allocators because we could no longer guarantee that critical sections be short. It is believed that user's needs in terms of resource semaphores will be much less demanding than the system's and that a library of allocators could be established for users which would satisfy almost all user demands. In exceptional cases there would be external procedures so that allocators written or approved by installation systems programmers could be added to the library.

A hardware solution to the problem of mutual exclusion

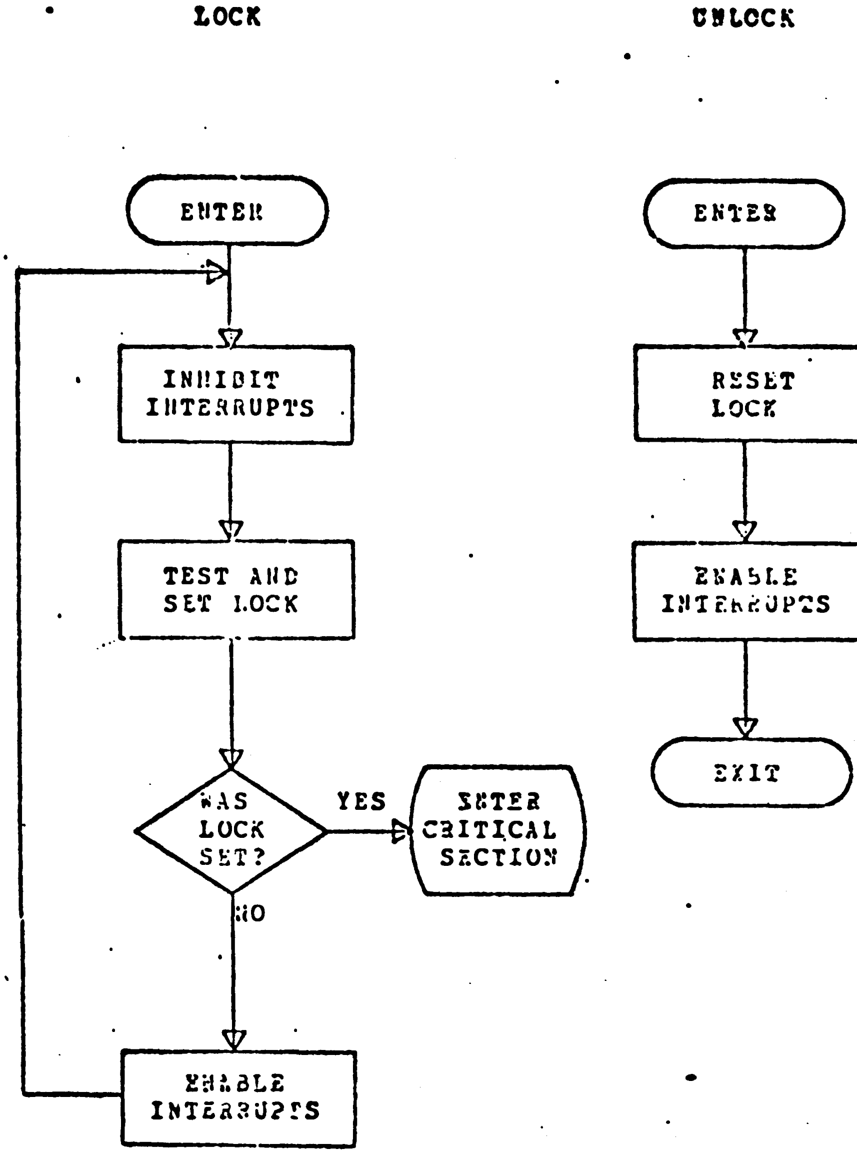


Figure 2.6 Flowcharts of a busy wait implementation of lock and unlock. The test and set operation is indivisible and sets the lock only if the lock was originally reset (indicating no process is in its critical section with respect to the lock).

might be to have a limited number of indivisible operations which are somewhat more powerful than the test and set instruction on the IBM System/360. Two such instructions for example might insert an element on a FIFO list and remove an element from a FIFO list where the list would have the simplest possible format and the operands would be the address of the listhead and for the insert operation the address of the element being inserted. When one processor executes an insert or remove on a listhead no other processor could execute one of these operations on the same listhead until the first processor completed its instruction cycle. These operations could then be used to implement higher level lock and unlock operations which would incorporate waiting lists of blocked processes. More research is needed in the area of machine operations for mutual exclusion.

The question in terms of implementing REQUEST and RELEASE on a conventional multiprocessor computer system remains one of whether to use a busy wait implementation of lock and unlock to protect the integrity of the resource semaphore or to use an idle wait implementation. An idle wait would mean that the resource data structure would have to be augmented by another list whose elements would be those processes waiting to access the rest of the resource semaphore data base. This would alter our definition of a logical resource slightly in that a process could be blocked and waiting for a resource (on the resource waiting list) or could be blocked and waiting for access to a resource (on the lock list). This subtracts somewhat from the elegance of the resource semaphore concept.

There is also the question of whether an idle wait would indeed be more efficient than a busy wait. The idle wait requires two process switches and two list operations to implement. Thus the busy wait is more efficient in cases in which the number of instructions executed in the wait loop is less than the number of instructions needed to execute two process switches and two list operations. We see that if the amount of time spent in a critical section is comparable to the cost of implementing an idle wait, there is little or nothing to be gained by the more sophisticated locking mechanism.

All in all, there appears to be little to support the use of an idle wait implementation of lock and unlock for use in REQUEST and RELEASE. For the implementation of idle wait mutual exclusion in the operating system as a whole, REQUEST and RELEASE would then be used with a single resource (representing permission to enter a critical section) and an allocator which would give this resource out on a first come, first served basis. We would then maintain the convention that all logically blocked processes would be on the waiting list for some logical resource.

IMPLEMENTATIONS OF REQUEST AND RELEASE

Figure 2.7 gives a straightfoward implementation of REQUEST and RELEASE in the higher level systems implementation language BCPL/360 [Kel 70]. The BCPL implementation ignores

```
LET REQUEST(RS,DATA) = VALOF  || REQUEST RETURNS A VALUE
$1 CURRENTP.(RQST) := DATA     || STORE DATA PARAMETER IN PCB
  BLOCK(CURRENTP)              || REMOVE PROCESS FROM RDY LIST
   INSERT(RS.(WAIT),CURRENTP)  || PROCESS TO WAITING LIST
   $2 LET PCB=(RS.(ALLOC))(RS) || APPLY ALLOCATOR TO THE RS
      IF PCB=0 THEN BREAK      || NO ALLOCATION => EXIT LOOP
      WAKEUP(PCB)              || PROCESS TO READY LIST
   #2 REPEAT                   || REPEAT UNTIL NO ALLOCATION
   SCHEDULER()                 || RESUME A READY PROCESS
   RESULTIS CURRENTP.GRANT     || ALLOCATION ASSIGNED BY
#1                             ||    ALLOCATOR IS RETURN VALUE

LET RELEASE(RS,RCB) BE
$1 INSERT(RS.(RESQ),RCB)       || RCB TO AVAIL RESOURCE LIST
   $2 LET PCB=(RS.(ALLOC))(RS) || APPLY ALLOCATOR TO THE RS
      IF PCB=0 THEN BREAK      || NO ALLOCATION => EXIT LOOP
      WAKEUP(PCB)              || PROCESS TO READY LIST
   #2 REPEAT                   || REPEAT UNTIL NO ALLOCATION
   SCHEDULER()                 || RESUME A READY PROCESS
#1
```

Figure 2.7 A BCPL/360 implementation of REQUEST and RELEASE
A.(I) is a vector application, F(X) is a function application,
|| starts a comment field, and $,# are *begin*, *end* brackets

the problems of mutual exclusion mentioned above because it was tested and run in an environment which did not permit interruption of the virtual BCPL machine. The routines appear exactly as they did in the simulation program to be described in subsequent chapters. These primitives were used to operate on 17 distinct resource semaphores with 4 different allocators and 9 different queue handling routines. The amount of IBM System/360 code generated for the body of REQUEST is 180 bytes while support routines including a very simple scheduler take another 392 bytes. The most frequently used allocator takes 144 bytes and the queue handling routines consume another 384 bytes for a total of 1100 bytes or about 275 instructions.

These figures are inflated by a high degree of subroutinization (at 24 bytes per subroutine call) because the routines were written to optimize clarity rather than space or time. The figures also include the code for keeping track of statistics associated with the various queues such as current number of elements, maximum number of elements, total number of elements entering and leaving the queue, the time of the last entry or removal, and the element time product for computing average queue lengths.

The number of instructions executed in the REQUEST primitive assuming that only one allocation is made is approximately 500. Even in this crude higher level language implementation the execution time is probably short enough (about .5 milliseconds) to allow the whole primitive to be treated as a critical section. If we only protect the critical portions of the primitive, eliminate the statistics gathering, and optimize some of the code, then the critical sections may be much shorter.

An assembly language implementation of REQUEST and RELEASE was written for a multiprocessing environment with interrupt-inhibited busy wait implementation of lock and unlock. Except for the inclusion of mutual exclusion primitives it is a straightfoward translation of the higher level language algorithms of Fig. 2.7. The implementation has many critical sections which may be nested and which protect different data bases independently. The body of the REQUEST requires 182 bytes, queue handlers and mutual exclusion routines require 106 bytes and the basic allocator requires 58 bytes for a total of 446 bytes or about 110 instructions. The number of instructions executed assuming one allocation is 167.

The assembly language version is thus about 60% smaller and 66% faster than the higher level language version. These figures would show even more improvement if the whole primitive were treated as a critical section rather than protecting each individual critical section separately.

2.8 Examples of resource semaphores

The purpose of this section is to present several applications of resource semaphores. The description of the resource semaphore includes the structure of the two queues, their associated queue handlers, and the resource allocator. Examples from each of the applications areas distinguished in Section 2.2, namely mutual exclusion, resource allocation, and message and data passing are presented.

SELECTIVE MUTUAL EXCLUSION

The first problem is a modification of the simple mutual exclusion problem and is called selective mutual exclusion. The application concerns the mutual access to singly linked lists by a number of cooperating sequential processes. Consider three possible operations on lists:

(1) removal of an element

(2) insertion of an element

(3) searching (no insertion, removal, or modification)

It should be obvious that no harm can come from letting two or more processes search a list simultaneously since no changes are made to the list and consequeatly each process is independent of other processes searching the list.

When an element is removed, on the other hand, no parallel access may be permitted to the list because a process might base its decisions on an element that is no longer on the list. Figure 2.8 shows what might happen to a three element list if one process attempts to remove an element A and the other attempts to remove B. The resulting list containing B and C is in error because of the parallel access.

If one process is removing an element and another process is inserting an element, it may happen that the inserted element is made to follow an element that was subsequently removed. This sequence of events is illustrated in Fig. 2.9. If we allow simultaneous removal and searching it may happen that the element being examined by the searching process is removed. The element's forward pointer is no longer necessarily valid so it may follow an erroneous pointer to get to the next element.

Finally consider insertions. Two simultaneous insertions may result in the sequence of events shown in Fig. 2.10. Again a modification of the list takes place on the basis of obsolete information giving erroneous results. If we do insertions properly, however, it is possible to allow simultaneous insertion and searching of lists. Proper insertion means the modification of the forward pointer of the new element before modifying the forward pointer of the new element's predecessor. Assuming the modification of pointers is one indivisible operation (a store operation on most machines), then the list is properly formed at all times.

If the searching processes are independent of processes which insert and remove elements they should be indifferent to the current state of the list.

To summarize we see that the removal operation requires

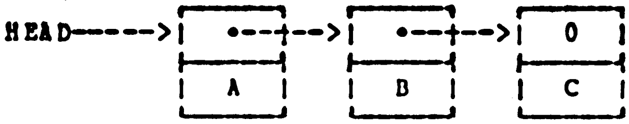


__Figure 2.8(a)__ Original list contains three elements. Process 1 finds element A first on the list and process 2 finds that B is the successor of A.

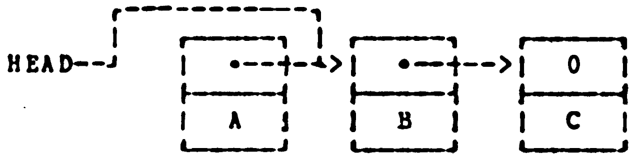


__Figure 2.8(b)__ Process 1 removes element A by having the list head pointer point to B.

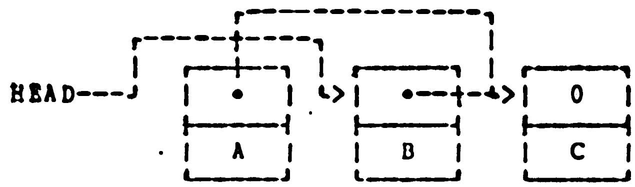


__Figure 2.8(c)__ Process 2 removes element B by making C the successor of A. The resulting list is in error because it consists of elements B and C.

__Figure 2.8__ Simultaneous deletion of two elements

```
                  +------------------------+
               +--+--+      +------+       | +------+
HEAD------>    |  .  |      |  ?   |       +>|  0   |
               +-----+      +------+         +------+
               |  A  |      |  B   |         |  C   |
               +-----+      +------+         +------+
```

Figure 2.9(a). Original list contains two elements A and C. Process 1 wishes to remove A and process 2 wishes to insert B. Both processes find that A is the first element on the list.

```
           +-------+--------------------------+
           |    +--+--+      +------+         | +------+
HEAD--+         |  .  |      |  ?   |         +>|  0   |
                +-----+      +------+           +------+
                |  A  |      |  B   |           |  C   |
                +-----+      +------+           +------+
```

Figure 2.9(b) Process 1 removes A by setting the list head to point to C.

```
           +----------------------------------+
           |    +-----+      +------+         | +------+
HEAD--+         |  .--+-->   |  .---+---------+>|  0   |
                +-----+      +------+           +------+
                |  A  |      |  B   |           |  C   |
                +-----+      +------+           +------+
```

Figure 2.9(c) Process 2, having already found A as the first element, makes B A's successor and C B's successor. The resulting list does not contain B.

Figure 2.9 Simultaneous insertion and deletion

```
         ┌-------------------------┐
         |   ┌─────┐    ┌─────┐    |  ┌─────┐
HEAD-----┘   │  ?  │    │  ?  │    └->│  0  │
             ├─────┤    ├─────┤       ├─────┤
             │  A  │    │  B  │       │  C  │
             └─────┘    └─────┘       └─────┘
```

__Figure 2.10(a)__ Original list contains one element. Process 1 wishes to insert A and process 2 wishes to insert B. Both processes find that C is the first element on the list.

```
               ┌---------------------┐
             ┌─┼───┐    ┌─────┐      |  ┌─────┐
HEAD-------->│  •  │    │  ?  │      └->│  0  │
             ├─────┤    ├─────┤         ├─────┤
             │  A  │    │  B  │         │  C  │
             └─────┘    └─────┘         └─────┘
```

__Figure 2.10(b)__ Process 1 makes A the new first element by modifying the list header and A's successor.

```
         ┌----------------┐
         |     ┌----------┼----------┐
         |   ┌─┼───┐  |  ┌─────┐    |  ┌─────┐
HEAD-----┘   │  •  │  └->│  •--┼----┼->│  0  │
             ├─────┤     ├─────┤       ├─────┤
             │  A  │     │  B  │       │  C  │
             └─────┘     └─────┘       └─────┘
```

__Figure 2.10(c)__ Process 2 makes B the new first element by modifying the list header and B's successor. The resulting list does not contain A.

__Figure 2.10__ Simultaneous insertions

exclusive access to a list. Insertion must be done exclusively of other insertions, but may be in parallel with searching, while searching may be performed in parallel with other searching. These results are summarized in Table 2.1. The problem is to allocate access to the list according to some prescribed strategy.

The difficulty lies in specifying a number of classes of critical sections. This is the reason we have elected to call this type of mutual exclusion <u>selective mutual exclusion</u>. With Dijkstra's semaphores, only one type of critical section exists. IBM's primitives, on the other hand, distinguish only two categories of critical sections; processes can request exclusive use or shared use of a resource. REQUEST and RELEASE, however, allow arbitrarily many categories of critical sections for the purpose of mutual exclusion as is shown by its application to this three category case.

The resource semaphore consists of the two lists and the allocator to be described. The DATA parameter of the REQUEST primitive is either R, I, or S indicating whether removal, insertion, or search access is requested. The resource list consists of three RCBs representing three different kinds of access to the list. Each RCB has a count field indicating the number of processes which have been granted access of that particular type. The allocator decides based on these usage counts and the processes on the waiting list whether to grant access to one of the processes.

If an allocation is possible, the appropriate count is

Table 2.1 List accessing rules

Process 2 \ Process 1	R	I	S
R	NO		
I	NO	NO	
S	NO	YES	YES

R: REMOVE ACCESS
I: INSERT ACCESS
S: SEARCH ACCESS

updated, the chosen PCB is removed from the waiting list and assigned the address of the RCB which corresponds to its request. The remove queue handler associated with the resource list, instead of removing one of the RCBs increments the access count of that particular RCB. Thus the number of RCBs on the resource list remains constant. The counts for the P and I RCBs have a range of zero or one, while the S RCB may take any non-negative number as a value. When a process releases an RCB to the resource semaphore, the insert queue handler associated with the resource queue, instead of changing the number of elements in the queue, merely decrements the total number of outstanding accesses for the type pointed to by the element (E) parameter.

The allocator can follow a number of strategies. One strategy would be to honor any allowable request. This means that R access might be postponed indefinitely if there is a high volume of shared usage. This is because requests for S access is always be granted in the absence of R access, but R access is only granted when there are no other processes accessing the list. Thus this algorithm would seriously prejudice the chances of those processes wishing to remove elements from the list.

Another scheduling algorithm might only grant S requests if there are no R requests pending. This means that when an R access request is made the request is not delayed by any possible future requests for S access. As soon as the current processes with S access relinquish their access rights, then the R access request would be granted. A similar strategy would be to service the waiting queue in strict FIFO order so that if the request of the process on the top of the list can be granted, it is, otherwise no allocation is possible. This last form of the allocator is flowcharted in Fig. 2.11.

RESOURCE ALLOCATION EXAMPLE

The second applications area as defined in Section 2.2 is resource allocation. The buffering problem is a very practical one which crops up in almost all modern operating systems. In its simplest form the problem consists of coordinating the activities of two processes which we will call the producer and the consumer. Each of the processes has access to a fixed number of buffers which can contain a given amount of data. The producer process cyclically fills empty buffers with data, while the consumer process cyclically empties full buffers. When there are no more empty buffers to fill, the producer must go to sleep and wait for an empty buffer. The consumer must go to sleep when there are no more full buffers.

The resource classes in this example are empty buffers and full buffers, so that there is one resource semaphore for each category of buffers. The elements of the resource list are the buffers themselves together with linking information. In the case of full buffers the ordering is important since the data usually represents consecutive records of a file of information. Hence the queue discipline for the full buffe

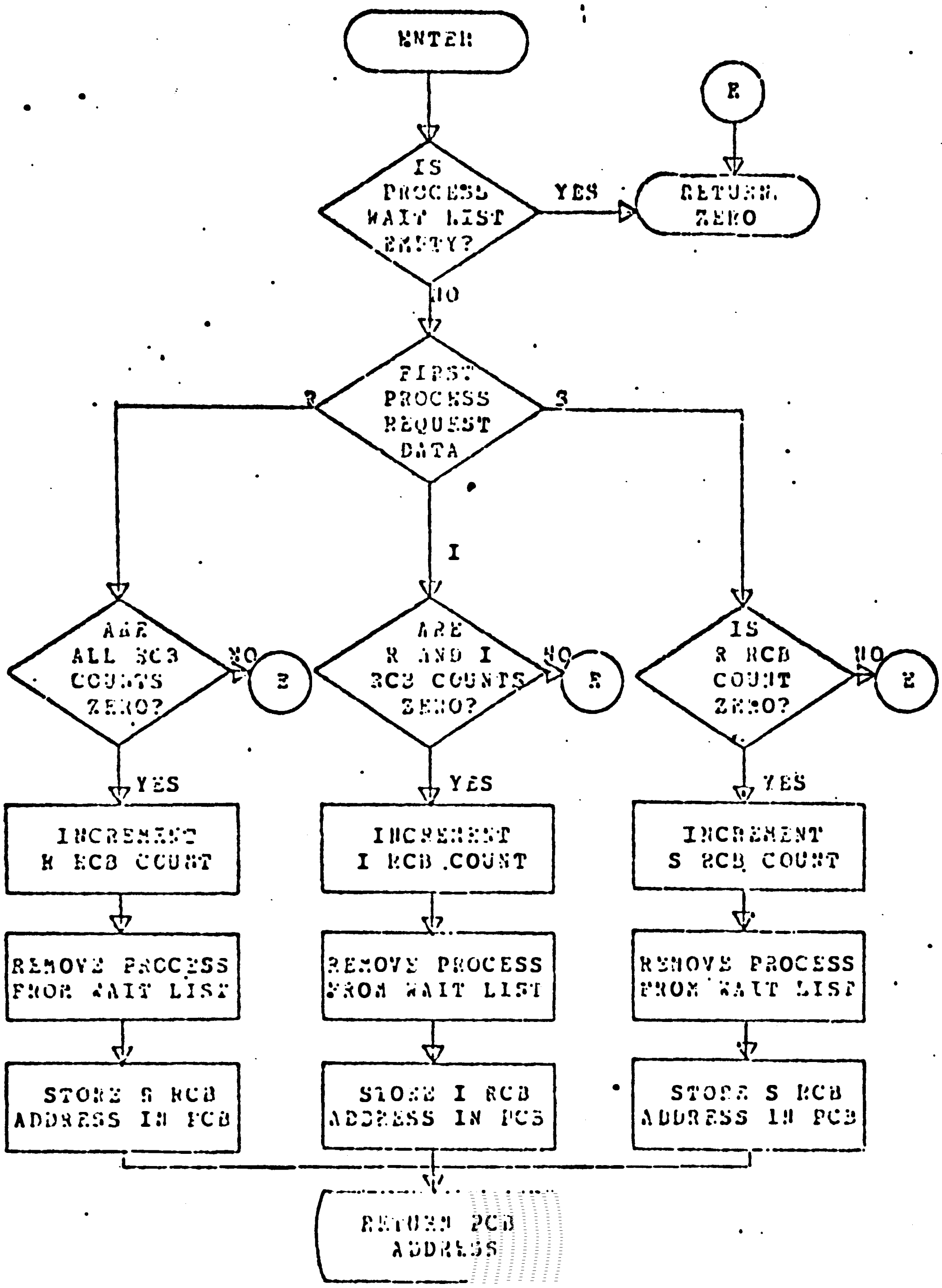


Figure 2.11 Flowchart of allocation for selective mutual exclusion resource semaphore

resource list must be strictly FIFO. In the case of one producer and one consumer, each of the process waiting lists contain zero or one process, so the queue discipline is again immaterial. If many producer-consumer pairs are using the same buffer pool, the number of processes waiting for an empty buffer could be greater than one so that a process list ranked according to priority might be appropriate. The case of multiple producer-consumer pairs would also necessitate multiple full buffer semaphores, each representing a single flow of data between two processes.

The allocator for both the full and empty buffer resource semaphores is the match allocator referred to in Section 2.5 and is flowcharted in Fig. 2.12. An allocation is made if and only if both the resource list and the waiting process list are non-empty. When this is the case the first RCB is removed from the resource list and the first PCB is removed from the waiting list. The address of the RCB is stored in the PCB of the requesting process and the address of the PCB is returned by the allocator. The REQUEST primitive is not qualified in any way by the DATA parameter, and has as its value the address of an empty or a full buffer. The producer then repeats the following sequence of operations:

```
B := REQUEST(EMPTY_BUFFER_SEMAPHORE)
Fill buffer at B
RELEASE(FULL_BUFFER_SEMAPHORE,B)
```

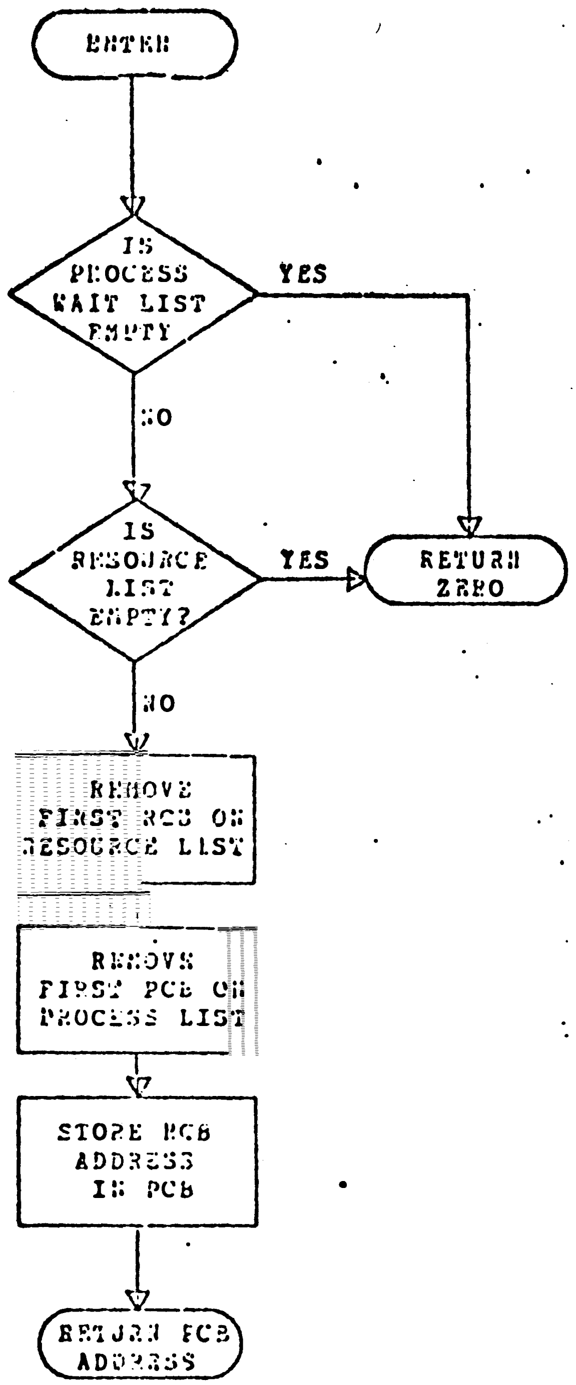


Figure 2.12 Flowchart of Basic Allocator

The consumer repeats these operations:

B := REQUEST(FULL_BUFFER_SEMAPHORE)

Empty buffer at B

RELEASE(EMPTY_BUFFER_SEMAPHORE,B)

The resource semaphore primitives also suppress all of the list processing details of buffer handling and make them invisible to the user. All other known synchronization primitives require that the user perform many straightfoward list operations and mutual exclusion at the expense of programmer effort and program clarity.

MESSAGE AND DATA PASSING

The third applications area mentioned in Section 2.2 is that of message and data passing between processes. The basic problem is to send to a particular process a message indicating that a certain event has occurred such as the completion of a task. The messages passed between processes may be fixed or variable length. Message passing is an example of producer-consumer resource sharing which uses explicit communication between processes because the sender of a message names the receiver explicitly.

The resource control blocks associated with the resource semaphore are the message buffers together with the appropriate linking fields. If the semaphore is associated with more than one receiver then the RCB must also contain some identification of the process to which the message is directed. Both the resource list and the waiting process list may be FIFO. In fact the waiting process list in the case of a message semaphore serving one process contains either zero or one process.

The process wishing to send a message to another process RELEASEs to a resource semaphore an address of a message (MCB). A process wishing to receive a message REQUESTs a message from the particular semaphore where it expects the messages to arrive. The DATA parameter in this case could be used to indicate a particular subclass of messages or whether the request for a message is conditional or not.

The message handling allocator in the simplest case is the basic allocator which is flowcharted in Fig. 2.12 and is the same one used in the buffering example. For the centralized message handling facility in which more than one process can wait for the same semaphore, the allocator must take the form of the previously defined match allocator whose flowchart appears in Fig. 2.13. For each message on the resource list and for each process on the waiting list the allocator checks whether the message is being directed to the process. If it finds such a pair, it removes the message and the process from their lists and stores the message address in the PCB.

This implementation of message passing compares favorably with Brinch Hansen's message handling primitives of Section 2.3. In Brinch Hansen's scheme we are forced to have exactly two queues associated with each process, namely a queue of messages and a queue of answers. The REQUEST and RELEASE

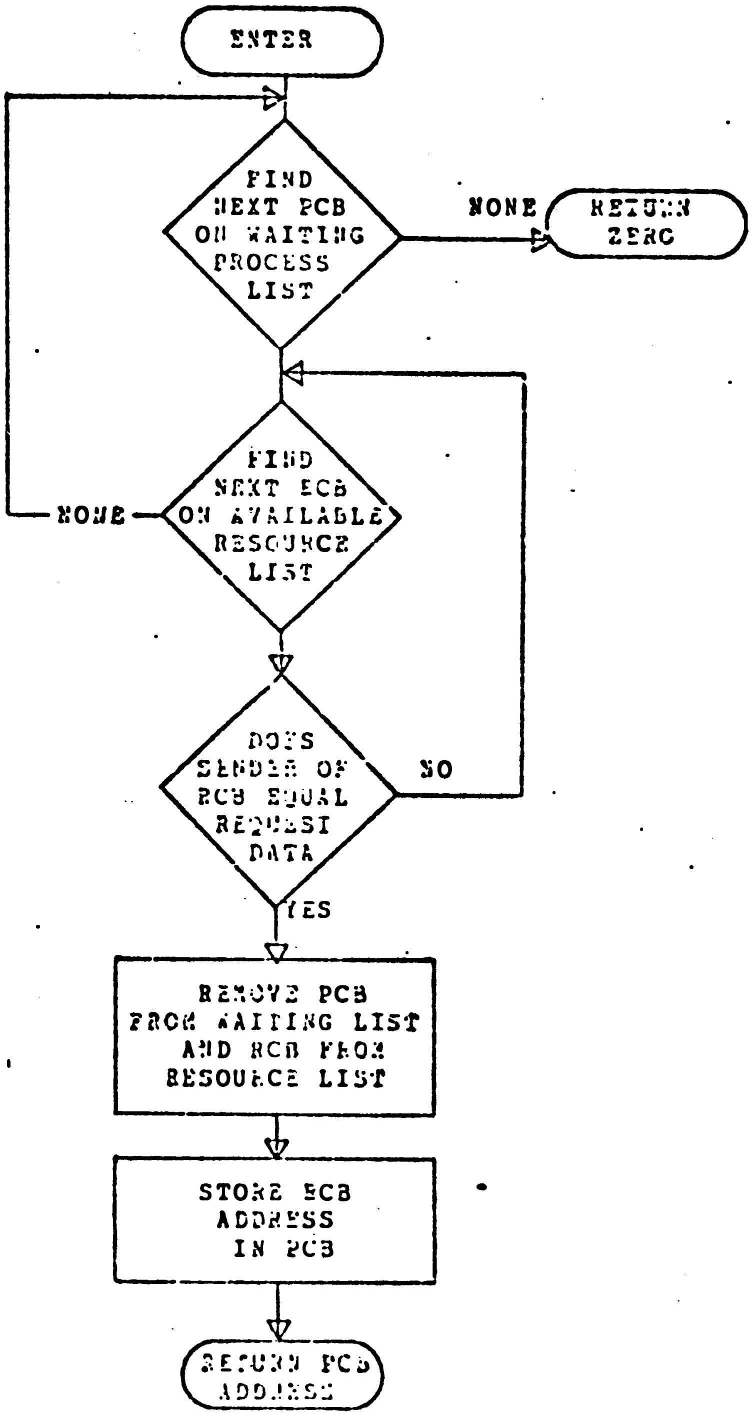


Figure 2.13 Flowchart of Match Allocator

implementation makes no distinction between messages and answers (although this could easily be done by storing that information in the MCBs). Thus a queue of messages may be associated with any number of processes. Brinch Hansen's primitives provide a good deal of checking of the validity of the conversations. This can also be done by the allocator of the message resource semaphore if necessary. The important point is that this checking is optional from semaphore to semaphore. Also, the way in which a process could service its queue would be variable where it is fixed by Brinch Hansen's primitives.

2.9 Evaluation of the resource semaphore concept

The purpose of this chapter has been to show similarities among a wide variety of synchronization and resource allocation problems and to introduce two primitives for their solution. There are several reasons for wishing to unify the various synchronization problems. First and most important, it is an attempt to make complex software systems easier to write, understand, and modify. Second, by centralizing the most fundamental operations associated with synchronization, we eliminate the effort of duplicating the common parts of these operations in many different places. Finally the convention that the only time a process may become blocked is when it executes a REQUEST is useful to the understanding and the analysis of the system.

Let us evaluate REQUEST and RELEASE in terms of three sets of primitives mentioned in Section 2.3 (Dijkstra's, IBM's and Brinch Hansen's). It is believed that these primitives provide a good cross section of existing synchronization techniques because most others are fairly closely related. The evaluation will be based on three criteria:

(1) power and generality

(2) simplicity and ease of use

(3) run-time efficiency

The three preceeding examples of resource semaphores were meant to demonstrate the power and generality of REQUEST and RELEASE. It is quite straightfoward to implement the existin primitives presented using REQUEST and RELEASE. Hence REQUEST and RELEASE can provide all of the synchronization services that were previously provided by a number of specialized primitives. In addition to being able to mimic the services provided by other primitives, REQUEST and RELEASE are capable of providing services which were awkward using other primitives. Selective mutual exclusion is an example of this. Finally resource semaphores allow systems designers to solve resource allocation problems and relate resource allocation in a precise way to process synchronization.

The resource semaphore primitives provide part of the framework for talking about and writing systems on a more abstract level than was heretofore possible. It allows the designer to specify with a higher level language a society of cooperating sequential processes and their interactions without being concerned initially with details such as how buffers or processes are going to be linked together, what allocation strategy is used, and how mutual exclusion is handled.

All the designer need know about at the most abstract level is that at a given point in the program a process executing the program needs a logical resource and without that resource it will be necessary for the process to go to sleep until the resource becomes available. If each of a number of programmers has to write his own individually optimized version of inserting and removing resources and processes from lists, then each must re-solve the problems of mutual exclusion to the various data structures. Experience with higher level languages for writing systems [Cor 69] suggests that although some sections of code may be made very efficient by this approach, the system can be optimized better on the whole by the ease of understanding and uniformity provided by primitives such as REQUEST and RELEASE.

The third criterion of evaluation, run-time efficiency, was not a major consideration in the design and implementation of the resource semaphore primitives, but we will argue that they are not a great deal less efficient than the other primitives. First it should be stated that because of the well recognized tradeoff between generality and speed of execution, it is unreasonable to ask that REQUEST and RELEASE be faster or even as fast as the less general sets of primitives. A second observation is that accurate comparisons of efficiency are very difficult to make. This is because implementations are on different machines and because statistics on particular implementations are difficult or impossible to obtain. This particular comparison is based on statistics gathered on the two sets of IBM primitives run on a 360/65 and Brinch Hansen's data [Bri 69] given for his primitives.

Brinch Hansen claims that the execution time of a send message/wait answer pair of instructions is 1 millisecond on the RC 4000 computer with a basic instruction time of 4 microseconds. This means that about 250 instructions are executed for the two primitives. The wait message/send answer pair also require 1 millisecond to execute. IBM's WAIT/POST pair requires an execution time of approximately .3 milliseconds on the 360/65 whose average instruction time is about 1.2 microseconds for about 250 instructions executed. The ENQ/DEQ pair required about 5 milliseconds or about 4,000 instructions executed. This figure for ENQ/DEQ seems to be extraordinarily high, but can be partially explained by the fact that OS/360 uses a centralized queue for all mutual exclusion for all processes. Each ENQ/DEQ has as parameters an eight character major queue name and a variable sized minor queue name. Several tests showed the number of elements on this list to be between 30 and 50.

Comparing these figures with REQUEST and RELEASE, we found in Section 2.7 that the BCPL/360 implementation of REQUEST required the execution of about 500 machine instructions. Since RELEASE is somewhat simpler than REQUEST we can use this as an upper bound for RELEASE for a combined figure of about 1000 instructions. The AL/360 version which implemented REQUEST and RELEASE with the mutual exclusion for the most general environment required the execution of 308 instructions for the two primitives.

The conclusion one draws from these figures is that even subject to errors of a factor of 2 or 3, the primitives proposed are sufficiently efficient to warrant their consideration as an alternative to other primitives in the implementation of operating systems.

CHAPTER 3. SIMULATION OF OPERATING SYSTEMS

The purpose of this chapter is to show how simulation can be used as an integral part of the design and implementation of operating systems. At every stage of the design and implementation procedure a simulation program is the realization of the current state of the work and provides feedback on the performance of the final system based on the design. As the operating system design evolves, so does the simulation of that design. Each successive simulation is more and more detailed until the last step becomes the operating system. The operating system consists almost exclusively of code used in previous simulations.

The motivation for this idea is derived from a paper by Zurcher and Randell [Zur 68] who suggested the principle of simulations becoming operating systems, but failed to develop the concept further. This chapter attempts to show the merit of their original idea and present some of the techniques and tools which make this approach both feasible and useful.

The first section provides some motivation and objectives for simulating in this manner. Section 3.2 discusses the distinction between event based and activity based simulation languages. Section 3.3 shows how activity based simulation languages together with the notion of a resource introduced in the last chapter provide an excellent framework for simulating an operating system. Section 3.4 describes various technique for implementing an evolutionary simulation as well as methods for simulating various hardware devices. Tho last section of the chapter considers the advantages and disadvantages of various languages for implementing an operating system in this way.

3.1 The role of simulation

There are inherent similarities between the role of a simulation program and the role of an operating system. Each must coordinate a number of interdependent activities over time. Each must keep track of a number of current activities (elements in a current events list in a simulation language and elements of a ready list in an operating system) and a number of future activities (future events and blocked processes). Each must have a mechanism for switching its attention from one activity to another (the synchronization module and the task scheduler). Finally each must accumulate statistics on the system constituted by the activities.

The following five objectives are distinguished for a simulation-implementation:

(1) Partial designs are executable.

(2) Performance measures provide feedback at each level.

(3) The system is documented at all levels.

(4) The simulation provides a basis for modifications.

(5) The simulation code becomes implementation code.

The basic idea is that a system design must evolve slowly starting with the most general ideas successively adding more and more details. At each stage the simulation is an executable representation of the partially completed design. The fundamental entity at each phase of the design is a user job and the main criterion for deciding whether a design is a good one is the efficiency of the system in processing user jobs. The measures of this efficiency may be different at different levels of simulation.

Executable partial designs are necessary in order to obtain some idea of final system performance before the entire system is implemented. Since the simulation is a realization of a design at a given level of detail it also provides excellent documentation of the system at that level because it is less ambiguous than English language documentation. Furthermore if a good higher level language is chosen for the simulation then this form of documentation can be highly readable. This is in contrast to documentation with assembly language or languages such as APL [Ive 62] which though precise are difficult to understand.

The fourth objective is that the simulation provides the basis for modifications to the system. The simulation should allow variability of the hardware configuration, the user job mix, and operating system software including interrupt handlers, schedulers, and allocators. Furthermore this ability should be available at each level of detail so that the feedback provided may be used to change the design and the simulation.

The last objective is that the code used in the simulation becomes the code used in the implementation of the operating system. This idea has never been exploited nor has anyone explained how this might be accomplished. The motivation for doing this is clearly to avoid the duplication of effort required to implement an operating system and at the same time implement an equally complex simulation which performs almost the same functions. Furthermore as the simulation evolves there is no problem of establishing its validity since there are no inconsistencies between the simulation code and the implementation code. Much of this chapter is devoted to demonstrating the feasibility of this approach and to describe the principles involved in realizing this objective.

The feasibility of a simulation-implementation is strongly dependent on the run time structure of the simulation language which is chosen. The next section defines two existing run time structures for simulation languages and shows how our concept of a simulation-implementation depends critically on the activity based languages.

3.2 Event and activity based simulation languages

EVENT BASED SIMULATION

The basic components of an event based simulation language are shown in Fig. 3.1. They are:

(1) synchronization mechanism (SYNCH)

(2) event subroutines (EVTi)

(3) simulation clock (CLOCK)

(4) system status variables

(5) event list containing event-notices (EiN)

In an event based system anything which changes the status of the system is called an event. Corresponding to each of a fixed number of events is a subroutine which changes the values of one or more variables representing the system being simulated or which schedules other events. Events take no simulated time.

While the event subroutines are a static representation of each event, the event list is a dynamic schedule of those events which are to happen during the execution of the simulation. Each element of the event list is called an event-notice and consists of two components, the name of the event which is to happen and the simulation time at which the event is to occur. The event list is ordered according to increasing time with the first element on the list representing the next most imminent event.

The synchronization mechanism removes the next most imminent event-notice from the event list, increments the

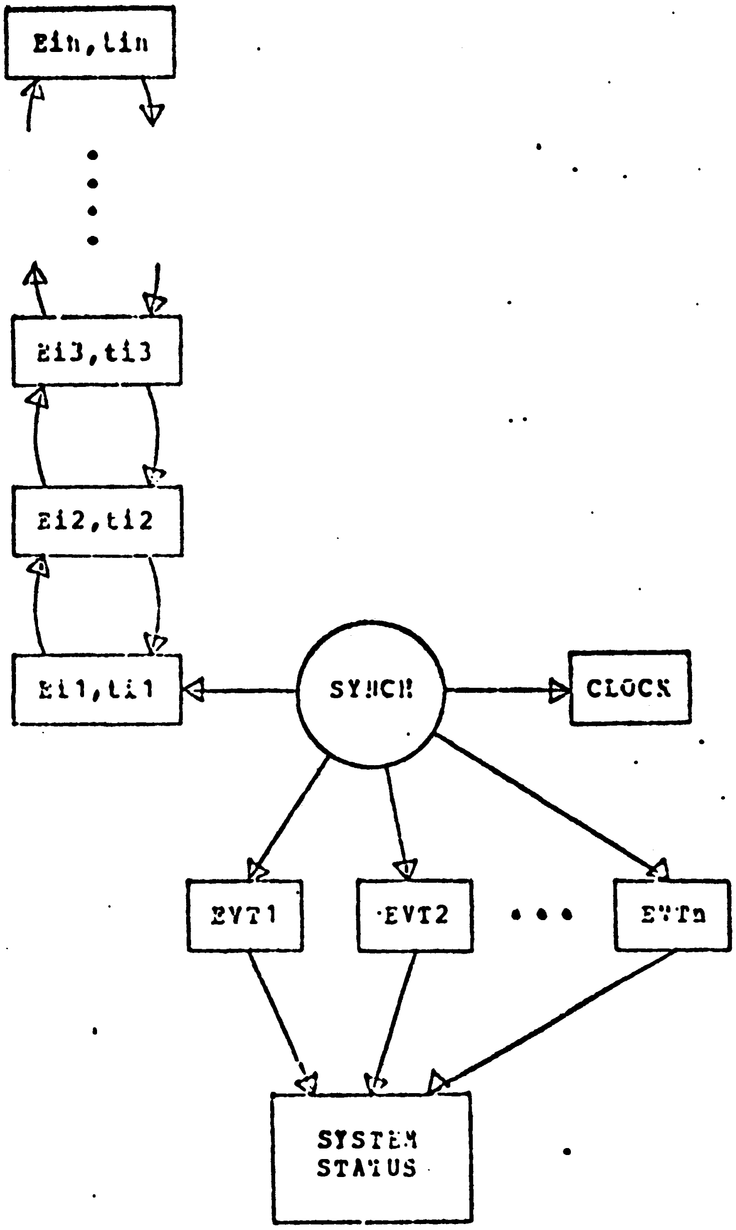


Figure 3.1 Event based simulation

simulation clock to tho time specified by the event element, and then transfers control to the event subroutine named by the event-notice. When the event subroutine returns control to the synchronization mechanism, the whole process is repeated until some event signals the end of the simulation.

Usually a distinction is made between two kinds of events. Those that are scheduled by a mechanism outside the system are called <u>exogenous events</u>. Those events which are scheduled during the execution of other events are called <u>endogenous events</u>. Scheduling operations have the form, say:

<u>Cause</u> <event name> <u>at</u> <time>

where <event name> is the name of some event subroutine and <time> is the simulated time at which the event is to happen. The actual time parameter may be an arbitrarily complex function and may depend on the current simulation time.

Most event based simulation languages provide an extensive set of list processing operations which allow the system status to consist of an arbitrary number of queues with many different queue disciplines. Primitives are provided to insert and remove elements from these queues, search them, and test them for emptiness.

The two most well known examples of event based simulation languages are SIMSCRIPT [Mar 63] and GPSS [Efr 64]. Although the syntax of these two languages is very different, they both have the structure outlined in Fig. 3.1. They differ by the fact that in GPSS all the event subroutines are defined by the system so that the user may only specify the sequencing of these events. In SIMSCRIPT, on the other hand, all the event subroutines are defined by the user. GPSS is attractive to some users because it is based on a flowchart language which makes the specification of problems in the language somewhat easier.

ACTIVITY BASED SIMULATION

The basic components of an activity based simulation appear in Fig. 3.2. They are:

(1) current activities list

(2) future activities list

(3) system status

(4) synchronization mechanism

(5) simulation clock

In the activity based simulation the fundamental component in the system is an <u>activity</u> which represents a sequence of events related to a particular entity of the system. In an event based system each subroutine represents an event that takes place instantaneously in simulated time. In activity based systems each procedure represents an activity which takes place over a finite amount of time. Since many activities are occurring simultaneously the execution of procedures representing activities must be interleaved and the timing constraints must be specified explicitly as part of the activity.

This interleaved execution of activities is achieved by a coroutine structure such that each time an activity is

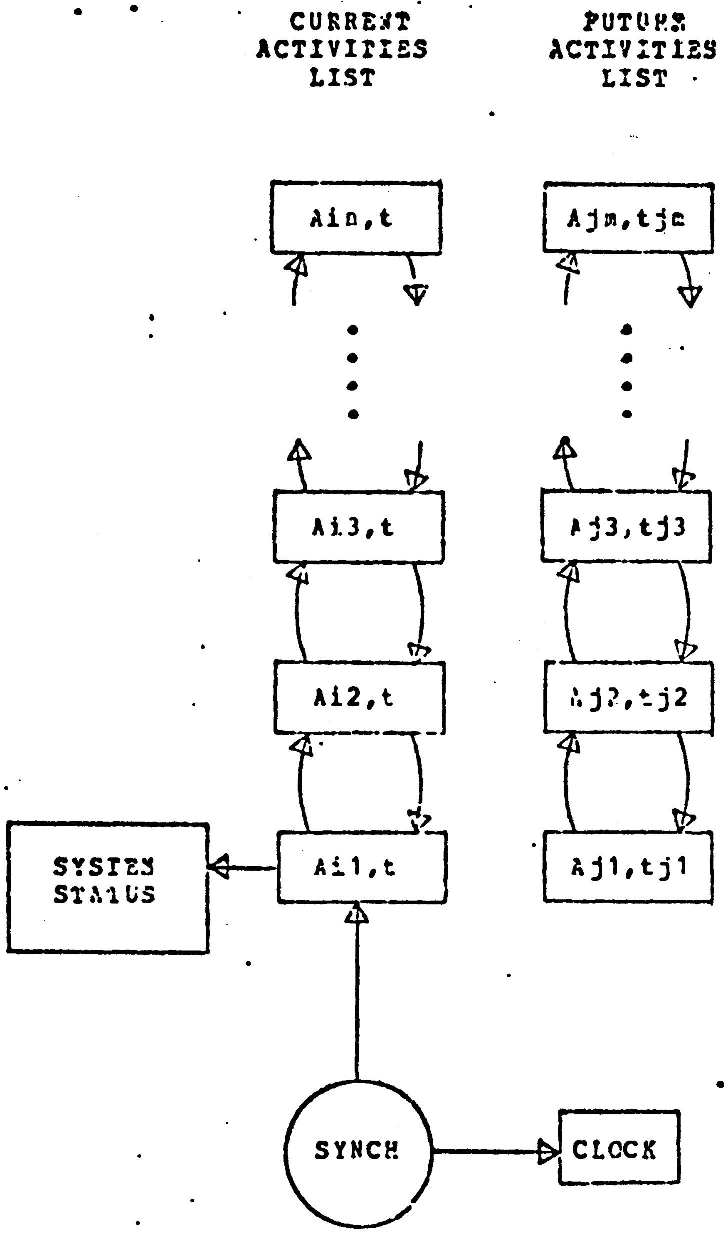


Figure 3.2 Activity based simulation

resumed, it is continued until a delay is indicated. The coroutine then becomes dormant in simulated time until the simulation clock is incremented to the end of the specified inactive period.

The current activities list consists of all activities which have active phases scheduled for the current simulation clock time. Each element, called an <u>activity-notice</u>, represents a particular activation of an activity and consists of three components. The first component is the name of the activity, the second component is an entry point indicating the point at which the activity is to be resumed, and the third component specifies the time at which the activity is to be resumed. The current activities list is distinguished by elements which have the same activation time, namely the current simulation time. The future activities list contains elements representing activities which will be resumed in the future and is ordered according to the time of the activation.

Many activity-notices may exist for the same activity. For example, in the simulation of a gas station we might have a customer activity and an attendant activity. Each individual customer and attendant (procedures) would be represented by a different activity-notice (process), several of which might be active in the system at the same time.

As in event based systems, the system status consists of state variables and queues of various entities of the simulation. The simulation clock keeps track of the current simulation time. The synchronization mechanism coordinates the various activities. If the current activities list is not empty, the first activity on the list is resumed at the specified entry point. This activity is executed until it invokes a scheduling primitive which removes it from the current events list and places it on the future events list. Control is then returned to the synchronization mechanism which chooses the next activity to be resumed.

If the current activity list is empty, the synchronization mechanism increments the simulation clock to the time of the next most imminent activity or activities as determined by the time field of the first activity-notice on the future activities list. Next, all the activity-notices whose time field equals the current simulation time are transferred to the current activities list. Each of these activities are resumed in turn until the list is again empty and then the process is repeated.

Scheduling operations which can be executed by the activities could include the following:

(1) <u>delay</u> <time>

(2) <u>reschedule</u> <u>at</u> <time>

(3) <u>reschedule</u> <time> <u>[after|before]</u> <activity>

Statements of the first type would remove the activity-notices from the current activity list, set the entry point to the next statement in the activity, set the reactivation time to current simulation time + <time> and insert the activity-notice in the proper place on the future activities list. Whereas the time specified in the first statement is relative to the current time, the time in the second statement would be regarded as an absolute time. Statements of the third type would permit scheduling relative to other activities in the simulation. Statements must also be provided to create new activity-notices of a given type and to destroy activity-notices.

The activity-notices on the future activities list can be thought of as instances of activities whose logical progress is not permissible at the current simulation time, while those on the current activities list represent activities whose logical progress is permissible at the current simulation time. It is up to the activities themselves to decide when they can no longer proceed. However, once on the future activities list they are activated automatically without intervention from other activities at the proper simulation time.

One advantage of an activity based simulation over an event based simulation is the ability to group together events which are related to the same entity, but which are separated in time. Let us again use the example of a gas station attendant. Suppose we wish to simulate the attendant as performing two tasks for each customer, namely filling the tank with gas and taking payment, each of which takes a certain amount of time. In an event based system an attendant could be implemented as four subroutines representing respectively the event of starting to fill the tank (at which time the next event would be scheduled), the event of completing the fillup, the event of starting to take payment, and the event of completing taking payment. In an activity based system an attendant could be represented by a single activity procedure which would specify the appropriate delays required while filling a gas tank and while accepting payment.

Examples of activity based simulation languages are SIMULA [Dah 66] which is based on ALGOL and SOL [Knu 64]. Both of these languages are described as general purpose simulation languages, and are therefore not completely appropriate for the simulation-implementation techniques to be described. Language considerations are discussed in Section 3.5

3.3 Activity based operating system simulation

The relationship between activity based simulation and operating systems is shown in Fig. 3.3. The operating system simulation will have the following components:

(1) ready list

(2) delay list

(3) resource lists

(4) operating system status

(5) process scheduler

(6) simulation clock

(7) idle process

The elements of each of the lists are representations of processes (PCBs). The ready list contains all those processes whose progress is logically permissible. Each PCB on the list contains an instruction pointer which indicates the point in the program at which the process should be resumed. The delay list contains PCBs which additionally contain a simulation time at which the process is to be resumed. The delay list corresponds to the future activities list. The PCBs on the delay list represent those processes whose logical progress is not permissible, but whose reason for blocking is unknown at the level of detail being simulated. PCBs on the resource lists represent processes waiting for particular resources.

The operating system status contains those state variables which constitute the status of the operating system such as resource queues, utilization statistics, and states of

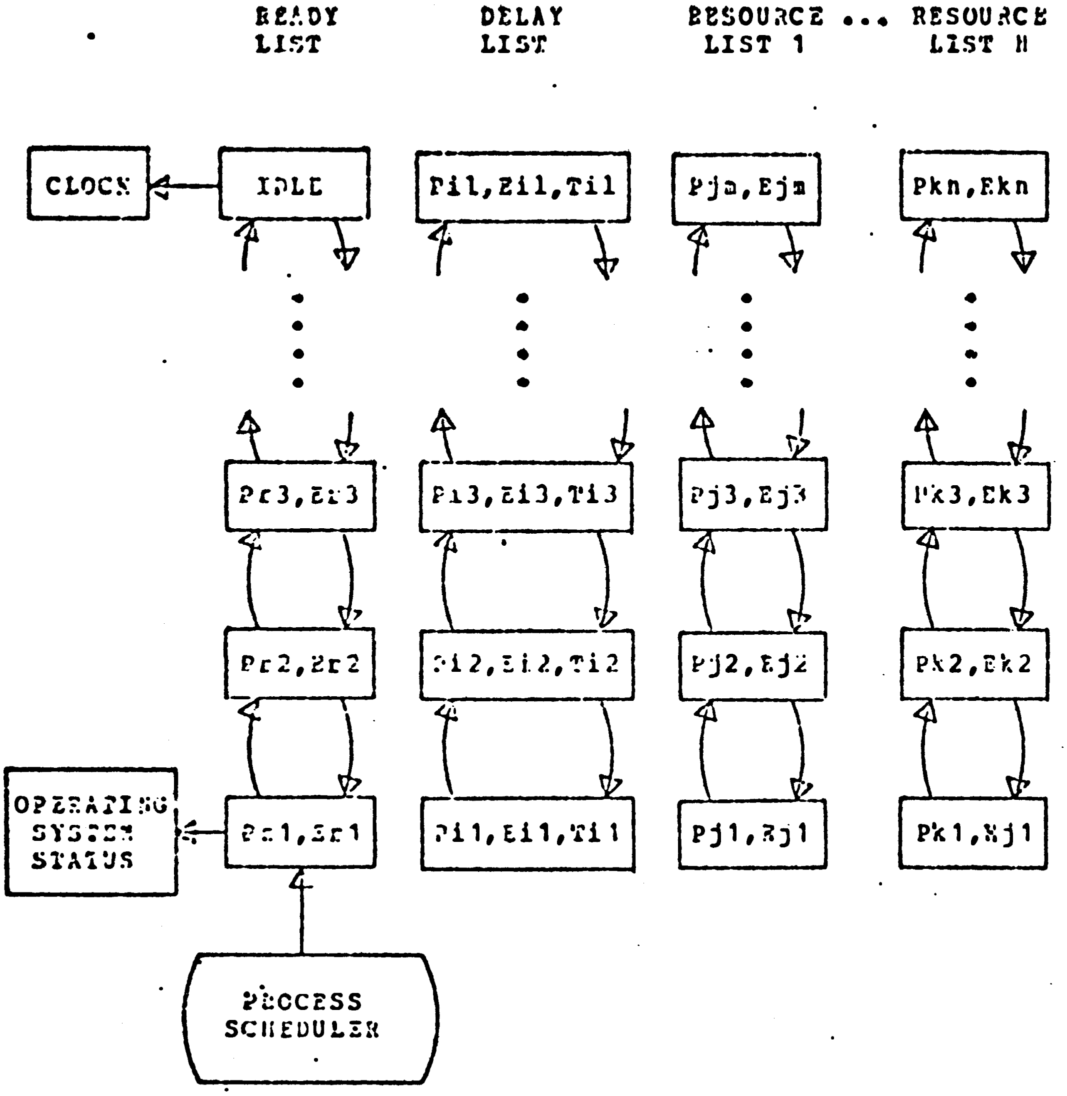


__Figure 3.3__ Operating system simulation

P - process
E - entry point
T - reactivation time

peripheral devices. The process scheduler takes the place of the synchronization mechanism and merely resumes one of the processes on the ready list. The PCB for the idle process is always the last element on the ready list. When the ready list is otherwise empty, the idle process is resumed. The idle process increments the simulation clock to the next most imminent event as represented by the first element on the delay list. It then transfers from the delay list all those processes to the ready list whose reactivation time is equal to the current simulation time. The process scheduler then resumes the processes on the ready list until it is empty except for the idle process. Then the whole procedure is repeated.

A process may be removed from the ready list and placed on a resource list when it performs a REQUEST primitive operation. Thus the resource lists are none other than the resource lists associated with each resource semaphore as described in the last chapter. When a RELEASE primitive operation is executed, a process may be transferred from a resource list to the ready list. Furthermore, since the delay list can also be incorporated into a resource semaphore, REQUEST and RELEASE are the only two operations which invoke scheduling activities so that the only changes to the ready or resource lists occur during the execution of one of these operations.

The power of the resource semaphore concept demonstrated by the fact that even the delay list implemented by a resource semaphore called, say, DELAY. The DATA paramater of the REQUEST is used to specify the amount of simulated time a process is to be delayed. The action resulting from the execution of a

REQUEST(DELAY,TIME)

is to place the invoking process on the delay list with a reactivation time of CURRENTTIME + TIME. The idle process would then execute a

RELEASE(DELAY,DUMMY)

after incrementing the simulation clock to the next event time. The allocator for the DELAY resource semaphore allocates a process if and only if the reactivation time specified as the request data is equal to the current simulation time. In this way, one or more processes would be transferred to the ready list for activation.

CORRESPONDENCE TO REAL SYSTEMS

The point of having a delay list in the simulation of an operating system is to be able to simulate software modules for which the details are unknown. It may be known that a given execution phase will take about a certain amount of time, but the way it interacts with the rest of the system and the number and type of system resources used is either unknown or as yet unspecified.

As the system evolves, the unspecified delays are replaced by specified delays. In other words, processes which once waited for a given amount of simulated time to pass request resources at more detailed levels which may cause the previously unspecified delays. In early simulations most of the processes are waiting on the delay queue, but as the system evolves, more resources are introduced and more processes are waiting for specific resources. The issue of making a correspondence between the simulated time of some processes and the real time of others is discussed in the next section.

In what sense is this a simulation of an operating system rather than an operating system implementation? The delay list, the simulation clock, and the idle process represent the last vestiges of any aspects of simulation. When we specify each of the system's processes down to the last level of detail there is no longer a need for the delay list since no processes requests the DELAY resource semaphore. Since no processes are then dependent on the simulation clock, that can also be eliminated. The final question is the nature of the idle process. When there are no processes left on the ready list, there is no useful work for the processor to do. Since there are no ready processes the only source of new work for the operating system is an external interrupt such as an I/O interrupt. Thus we may change the idle process to a process which simply loops, or to an enabled wait state. When an interrupt is taken, one or more processes is made ready by the processing of the interrupt.

Thus we see in simplified form how all aspects of the simulation are eventually eliminated. The activities of the general purpose simulation language become the systems and user processes of the operating system simulation and implementation. The synchronization mechanism takes the place of the process scheduler. The current activities list becomes the ready list. The simulation clock and the future activities list disappear. The resource semaphores and resource semaphore operations take care of all process-process interactions and resource allocation.

3.4 Principles of simulating operating systems

The last section provided a framework within which it would be possible to simulate an operating system and use the code of the simulation also in the implementation. This section presents some of the principles used in designing an evolutionary system in this way. The following techniques are described:

(1) simulating dummy software modules

(2) simulating dummy hardware peripherals

(3) simulating the CPU activity

(4) simulating interrupts

(5) simulating storages

In each case we show how the technique works as part of the simulation and what changes need to occur to make it part of the implementation.

DUMMY SOFTWARE

Early in the design stages of an operating system, the designers may have ideas about the overall structure of the system in terms of its major processes and their interrelationships without knowing the details of their implementation. In such cases it is useful to make use of dummy software in a simulation. A dummy software module is characterized by the fact that it takes (simulated) time, but it doesn't do anything useful. In other words it merely wai on the delay list for a given amount of time before returned to the ready list.

The amount of time spent on the delay list is an estimate of the amount of time the module would have taken from beginning to end in the final implementation. This estimate would include time spent, for example, waiting for other resources. In some cases these estimates can be make quite accurately, especially when it is known that the amount of interaction with other processes is small. In other cases these estimates may become little more than intelligent guesses. The problem of making estimates of time spent by dummy software is discussed later in the thesis.

Suppose we have a cyclic process of the form:

```
DO WHILE TRUE;
   REQUEST(DELAY,T);
   OTHER STATEMENTS;
   END;
```

where "OTHER STATEMENTS" represents what is known about the process in terms of its global effects on the system and "REQUEST(DELAY,T)" represents unknown time dependent interactions with the system. The result of this process in the simulation is to execute the OTHER STATEMENTS every T units of simulated time if T is a constant. When T is a variable, the OTHER STATEMENTS are executed at unequal intervals. T could depend, for example, on the characteristics of user jobs in the system or on the characteristics of various hardware devices.

When the time comes to specify what has been previously unspecified in this process, we can merely replace the delay statement by a subroutine call as follows:

```
DO WHILE TRUE:
   CALL MOREDETAIL;
   OTHER STATEMENTS;
   END;
```

where the subroutine MOREDETAIL expresses more explicitly the interactions with the rest of the system in terms of REQUESTs and RELEASEs. If the subroutine contains any delay statements, then we are still simulating, but at greater level of detail. If the subroutine contains no delay statements then there is no longer any aspect of simulation in the process (except perhaps to coordinate simulated time with real time) and process behavior is dependent only on its interaction with other processes in the system through the resource semaphores.

When real and simulated software are intermixed it is necessary to coordinate the simulated time spent by dummy modules with real time spent by real modules. This is accomplished by having the real modules execute a SPEND primitive to increment simulated time by an amount equal to its real execution time. A more detailed treatment of this problem is given in the subsection on simulating the CPU.

SIMULATING HARDWARE PERIPHERALS

In any operating system simulation it is desirable to be able to simulate hardware peripheral devices by internal processes. Consider the problem of communication between an internal process and an external device. In terms of messages we can distinguish four distinct phases for the conversation between the process and the device in the simplest case.

(1) internal process signals desire for an I/O operation

(2) external device receives signal and starts I/O

(3) external device signals end of I/O or error

(4) internal process receives completion signal

Once the internal process has signaled for an I/O operation to take place, it must wait for a completion signal before it can assume that the operation has terminated.

The external device can be simulated with two resource semaphores called respectively DEVICE and DEVICECOMPLETE. The internal process executes the two statements:

```
RELEASE(DEVICE,COMMAND)
RESULT := REQUEST(DEVICECOMPLETE,CURRENTP)
```

when it wants to execute an I/O operation on a particular device. The COMMAND RCB contains the information needed by the device to perform the operation such as whether it is an input or output command and the location to and from which the data is to be transferred.

The simulated I/O device consists of a cyclic process of the form:

```
DO WHILE TRUE;
   COMMAND := REQUEST(DEVICE);
   SIMULATE DATA TRANSFER;
   REQUEST(DELAY,XFERTIME);
   RELEASE(DEVICECOMPLETE,COMMAND);
   END;
```

The process is dormant on the DEVICE waiting process list until a COMMAND arrives in the DEVICE resource list. When this happens, the process is activated and the data transfer is simulated if data structures are set up to do so. Then the process is delayed for an amount of time equal to the time necessary to transfer the data in the final system implementation. Next, the completion of the I/O operation is signaled by releasing the command to the DEVICECOMPLETE resource semaphore. This awakens the process waiting for the completion of the I/O operation. Errors in the I/O operation can be signalled by placing the appropriate error codes in the command RCB.

Since several processes may be waiting for the completion of an I/O operation on any particular device, the allocator for the DEVICECOMPLETE resource semaphore must be a match allocator. The completion signal must wake up the one and only one process which initiated the I/O operation in the first place. For this reason the internal process sending the I/O command must specify the DATA parameter CURRENTP as a unique process identifier.

The simulation of peripheral devices represents one of the cases in which the delay is easy to estimate since it usually depends only on the hardware that is used. This permits a straightfoward means of simulating peripheral devices of varying speeds by simply changing the value of the XFERTIME parameter in the delay statement. Peripherals which are data storage devices present a somewhat more difficult problem. In addition to the data transfer time, there may be a so called access time representing the time it takes to mechanically move the data or read/write heads in preparation for data transfer, and a rotational delay time in the case of disks and drum memories. The problem is that the access times and rotational delays are not constant and depend on the current state of the device. The access time also depends on the organization of the data on the storage device.

One solution to this problem is to take the access times from a known distribution. In simulations one might also want to use worst case or best case access times. At more detailed levels of the simulation when alternative storage allocation strategies are being tested, it may be possible to compute for each I/O operation the exact access time depending on the state of the device and the location of the data.

When the internal process is to be replaced by the actual hardware peripheral device, the result of a RELEASE(DEVICE,COMMAND) statement must be to execute a machine instruction which starts the I/O operation with the appropriate parameters. On the IBM 360 series of machines it is the SIO instruction whose parameter specifies the I/O device and channel.

The REQUEST(DEVICECOMPLETE,CURRENTP) statement must result in the process being blocked until the interrupt corresponding to the I/O operation being initiated is processed. In addition it is necessary to specify the operands of the I/O operation in the form and location specified by the real machine organization. Some of these operations may have to be done in master mode. Since REQUEST and RELEASE are nucleus operations they would have the required supervisory capabilities.

One easy way to make the conversion from simulated to real mode is to maintain the same structure of resource semaphores and replace the delay statement in the simulated external process by an instruction sequence (a SIO and a WAIT macro on the IBM/360) which starts the I/O operation and then waits until it is completed. Under this arrangement there would probably be a pair of resource semaphores corresponding to each channel so that the process associated with each channel could be considered a software-hardware interface process. The allocator associated with the DEVICE resource semaphore determines the order in which commands are serviced. The internal process releases the logical I/O operations and the interface process interprets these commands and issues real I/O operations.

The disadvantage of this solution is that the interface process becomes logically blocked without execution of a REQUEST operation contrary to our proposed conventions. What is desired here would be hardware such that the channel processor would be sophisticated enough to be able to execute a RELEASE on a resource semaphore associated with the channel. If channels could communicate with internal processes in this way we would virtually eliminate the need for external interrupts. It has been suggested by Wirth and others that the interrupt is a "potential source of programming pitfalls and errors" [Wir 69], so there is some sentiment for eliminating interrupts anyway. The problem of simulating interrupts is explored later in this section.

SIMULATING THE CPU

There are two problems inherent in simulating the CPU of an operating system. The first is to account for the state of the CPU in terms of its registers, and the second is to account for the processing time in the simulation. If we assume that the operating system is being simulated on the same machine for which the implementation is being written, then the first consideration provides no <u>conceptual</u> difficulty. Since the simulation program is executing on the CPU and the simulation program becomes the implementation, the hardware registers of the CPU need not be modeled. In fact the simulated CPU is the real CPU from the very first level since it is needed to execute statements in the simulation language chosen.

The simulation of CPU time is a more difficult matter. A first attempt at solving this problem might be simply to execute a delay statement whenever it is wished to simulate a given amount of processor time. This solution would work, however, only if there were an unlimited number of processors since the processes executing the delay statement spend their time on the blocked list in parallel. As an example, if two processes executed the statement

REQUEST(DELAY,T)

at the same simulated time, say ST, to simulate T units of time on the processor, then both would be returned to the ready list at time ST + T. Furthermore, this does not prevent other processes from executing in the simulated time interval [ST,ST+T].

A second attempt at solution would be to have a process increment the simulation timer to indicate passage of time on the processor. Suppose we have an operation

SPEND(T)

which increments the simulation timer by T. This solution overcomes the objection that anything else can happen in the simulated time interval [ST,ST+T]. But this is too strong a solution because there may be processes on the delay list which are scheduled for activation in the interval [ST,ST+T].

The solution to this problem is to have the SPEND operation increment the clock only to the minimum of ST+T and the time of the next most imminent event on the delay list. In the former case the SPEND operation has been successfully completed. In the latter case the SPEND operation remains incomplete and a RELEASE of the DELAY resource semaphore may cause the executing process to be pre-empted by processes from the delay list. When the process is again resumed by the process scheduler, it effectively executes a call to SPEND for the amount of time remaining to be attributed to processor time.

If the simulation has developed to the point that time is still being simulated, but actual peripherals are being used instead of simulated peripherals, then a problem may arise as to the coordination of simulated time with real time. If an I/O operation takes X units of real time, we must insure that during that interval of time at most X units of simulated time passes. In other words, at the final stages of the simulation we require that simulated time and real time are approximately equal. Let us define the run time ratio, R, as:

$$R = \frac{\text{ELAPSED REAL TIME}}{\text{ELAPSED SIMULATION TIME}}$$

where ELAPSED REAL TIME is total CPU clock time for a given run and ELAPSED SIMULATION TIME is total simulation clock time. We require that as the simulation approaches the implementation the ELAPSED SIMULATION TIME remains roughly constant while the ELAPSED REAL TIME increases as code is added, and R approaches 1.

To establish the coordination between real and simulated time, the simulation clock is incremented by an amount equal to the running time of the code. This means that as long as any processes are still being simulated, all real processes must, on each activation, invoke the SPEND primitive to increment the simulation clock. When the simulation clock is eliminated, all the SPENDs are also eliminated and the simulation becomes an implementation. It is assumed that the real time required to execute the SPEND primitive is negligible compared with the body of the processes invoking it. The simulation time spent by real processes can be determined _a priori_ by inspecting the code, or by testing a hardware clock which is part of most computer hardware.

The run time ratio R should always be less than 1 in any sequence of simulation programs leading to an implementation because in any simulation there are always details of the implementation to be added, but very little to be deleted (only the SPEND primitives). This also means that the elapsed real time for any simulation should be larger than for any predecessor simulation. Hence if real time exceeds simulated time for any particular run we know that the estimated processing times in the simulation have been underestimated.

SIMULATING INTERRUPTS

We distinguish two kinds of interrupts, internal and external. Internal interrupts are those interrupts which are triggered by the activity of a process executing instructions on a CPU. The most common types of internal interrupts are supervisor calls which represent explicit demands for operating system services; timer interrupts which represent implicit demands by a process for system services; defective instruction interrupts; and defective data interrupts.

External interrupts are those generated by other processors. Examples are I/O interrupts (generated by the channel processor), interprocessor signals (initiated by other CPUs) and attention interrupts (generated by operators or users to get the attention of the CPU).

In most modern computers interrupts are handled in the following way:

(1) save stateword of executing process in fixed location

(2) get new stateword from another fixed location

(3) start executing at point indicated by new stateword

The basic result of an interrupt is thus to break off from one sequence of instructions and start on another.

First consider the simulation of internal interrupts. Supervisor call interrupts can be simulated and implemented using resource semaphores. In effect a supervisor call is a message to the system to perform the following service for the calling routine. Since the invoking process can not logically continue until the service is performed it may be considered logically blocked while the service is being performed. Thus we can consider a supervisor call as a

REQUEST(SVC,N)

where the DATA parameter of the REQUEST operation specifies the type of service requested. The allocator for the SVC resource would decode the request and perform the appropriate system routine. Alternatively the RELEASE primitive could be used to send a message to a systems process as described in Section 2.8 followed by a REQUEST to block the process until a completion signal is received. This would be the required procedure if the SVC routine generated requests for other resources.

Interrupts which make implicit demands on the system cannot be handled by an explicit mechanism such as the resource semaphore primitives. Many of these implicit type interrupts only come into play when execution of user jobs is introduced into the simulation. When user jobs are characterized as a sequence of time slices each of which is terminated by an internal interrupt, these interrupts can be simulated with a variable indicating which type of interrupt has occurred. Control is then passed to the appropriate interrupt handling routine which REQUESTs the appropriate resource semaphore if the process must become blocked. When real user jobs replace simulated user jobs the transfer of control takes place automatically through the normal hardware operation. The interrupt handlers remain unchanged.

The most common type of external interrupt is the I/O interrupt which was discussed in connection with simulating peripheral devices. Other types of external interrupts (attention interrupts and processor signals) can be simulated as messages from a source process to a high priority interrupt handling process. When the source process releases a message, the associated interrupt process is put on the ready list. After servicing the current interrupt the process goes to sleep until the next interrupt arrives.

SIMULATING STORAGES

Storages of computer systems exist on many hierarchical levels from the low volume, high access speed cache memories to the high volume, low access speed memories such as tapes and data cells. In between are core and integrated circuit memories, extended core memories, high speed drums and disks. Each of these memories can be simulated on a number of levels.

First it is necessary to decide on the basic unit of storage - bits, bytes, blocks, tracks, cylinders, volumes, etc. Then the simplest way to simulate a storage device is with a counter indicating the number of storage units in use or available. At early stages of the simulation we may wish to model a system with unlimited storage capacity to obtain a feeling for the performance of the system under these assumptions. Later we might want to consider behavior using fixed finite storage resources.

The latter case can be handled by resource semaphores which mimic Dijkstra's P and V with initial values of the semaphores equal to the total number of resources available. To keep the same framework for all levels of the simulation we can also handle the former case with resource semaphores by having an infinite (very large) initial inventory count. In either of these two cases the information in the storage is irrelevant.

At the next level of sophistication we may wish to add organization considerations, but still ignore the information being stored. This is done by upgrading the resource semaphore associated with the storage so that instead of a counter, the resource list contains information about the available storage. This list could be in the form of a list of variable or fixed sized storage control blocks. Alternatively, because of the flexibility of the insertion and removal routines, the list could take the form of a bit table which marks the empty and full blocks on a paging drum.

This level of modeling is appropriate when one wants to model the behavior of a system with an external storage device (one not directly addressable by the instructions of the processor) without having to perform time consuming I/O operations. This type of storage modeling is closely related to modeling peripheral processors as internal processes. Internal storage can also be modeled in this way, but no savings are realized by avoiding performing I/O operations.

When real user jobs and data are introduced and the operation of the system depends on the information in the storages, the real storage devices must be used to store this information. It may become necessary for reasons of efficiency to store the part of the resource semaphore data structure representing the organization of the storage on the storage device itself. The storage then ceases to be simulated and becomos part of the implementation.

It is possible to simulate information dependent peripherals in main or extended core memory, but because peripheral storage devices may contain so much more information than core memories, this technique becomes severely limited at a very early stage. However it may be worthwhile to do this to test some of the information dependent algorithms for file management.

SIMULATION OF USER JOBS

User jobs will be a constantly changing input to the sequence of simulations resulting in an implementation. The detail of the user job characterization reflects the detail of the operating system simulation. The characterization of a user job converges to a real user job (specifying the programs and data used by the job in some job control language) as the simulations converge to an implementation. Characteristics specified at one level are usually refinements of characteristics specified on previous levels. User job characterizations are discussed in detail in the next chapter.

3.5 Language considerations

Most of the general purpose activity based simulation languages are not appropriate for writing systems. Languages such as SOL and SIMULA do not have the facilities to create a sufficiently rich set of data structures for handling of different kinds of strings, queues, lists, stacks, and tables. Nor do these languages allow a flexible policy for the scheduling of current activities. The process scheduler (or synchronization mechanism) is fixed by the simulation language. Furthermore the general purpose languages do not provide the sequencing and statement grouping operations necessary for describing operating systems.

Another problem is that the general purpose languages fail to allow the execution of programs read in as data. This is an essential requirement for our purposes so that user programs may be executed by the operating system.

Since we would like to maintain the convenience of using a higher level language it would seem appropriate to chose one of the systems implementation languages and expand it to include the appropriate primitives for its use as a simulation as described above. Languages specifically designed for the implementation of systems are of two kinds, machine dependent languages such as PL360 [Wir 68] and BLISS [Wul 70] and the more machine independent languages such as BCPL [Ric 69] and SAL [Lan 69].

Fortunately the amount of effort required to convert one of these systems languages into a simulation language of the type described is relatively small. In fact the simulation is almost free since most of the data structures and algorithms are needed for the implementation of the operating system. The activities are described by process control blocks which are needed anyway. The delay queue is implemented as a resource semaphore which is part of the conceptual framework of the system nucleus. Basically the only construct which is exclusively simulation dependent is the clock. The only basic operations which are strictly simulation related are the operations on the clock such as the delay and spend statements.

It has been pointed out by Lang [Lan 70] that an important characteristic of a language for systems programs is that there be no run-time system of any kind loaded with the program. Examples of run-time routines are input/output packages and dynamic storage allocators. This requirement is equally important for the simulation of operating systems because it is necessary for the programmer to have complete control over all tho activities of the system.

Some further comments regarding systems languages are made in Section 4.2.

CHAPTER 4. A DESIGN METHODOLOGY

The dictionary defines "methodology" as the science of orderly arrangement. In the design and implementation of operating systems it has become a popular term to describe a particular strategy used in coming to grips with a software engineering problem of enormous complexity. The basic idea is that large systems cannot be built on an all-at-once basis so that it is necessary to design and build a system in a number of distinct stages. These stages are often called "levels" or "levels of abstraction."

There are many reasons for structuring the design of a complex software system. It tends to make systems written in this way easier to write, understand, debug or prove correct, and modify. Systems become unmanageable unless they are handled in a structured way because designers have certain limits of comprehension. When the complexity of the design exceeds these limits, the design tends to disintegrate. This phenomenon and its influence on organizations designing large software systems is explained in detail by M.E. Conway [Con 68]. If, on the other hand, each step of the design remains within the limits of one's comprehension, it will be tractable in terms of the four objectives mentioned above.

It is possible to distinguish two basically different design methodologies for the implementation of operating systems. We call them the bottom-up (or inside-out) approach and the _top-down_ (or outside-in) approach. Although these are called design methodologies, they may also refer to the way in which the system is built.

In the bottom-up approach one starts each level with a number of building blocks or primitive operations and then builds higher level operations from these. Starting with a very concrete real machine, one builds, in a number of distinct steps, a highly abstract, user oriented machine. The top-down approach starts with the specifications of an abstract machine and then works down specifying at each level how given components are implemented in terms of lower level components.

Dijkstra has used the bottom-up approach in the implementation of the THE-Multiprogramming System [Dij 68]. Parnas and Darringer [Par 67] and Zurcher and Randell [Zur 68] are advocates of the top-down approach. Section 4.1 presents a short summary of these methodologies. Section 4.2 outlines a methodology which is more appropriate for the implementation of systems using the basic principles of Chapters 2 and 3. Sections 4.3-4.5 provide more detailed explanations of the finer points of the methodology and Section 4.6 evaluates the methodology in terms of related work in this area.

4.1 Existing methodologies

DIJKSTRA'S METHODOLOGY

The key to understanding Dijkstra's implementation methodology is the concept of a "level of abstraction" or an "abstract machine." An abstract machine can be defined as a combination of hardware and software whose basic operations are designed to make it seem more useful and powerful than the real machine. In these terms an operating system can be defined as the software required to transform an unattractive, inefficient, difficult to use machine into a more attractive, efficient, user oriented machine.

The THE system was conceived and implemented in six distinct levels, each of which defined primitives or processes which constituted part of the abstract machine at that level. For example, at the first level the P and V operations are introduced to handle processor allocation. This introduces the abstraction of an infinite number of processors so that at subsequent levels processes can be written without regard for the number of processors in the system.

At the second hierarchical level, the segment controller process is added. The segment controller is responsible for defining the correspondence between the virtual address space and the real address space. In other words, if a segment of a user or system process is not in real memory and the process tries to access that segment, then the segment controller process is responsible for bringing that segment into real memory. This level introduces the abstraction that processes at succeeding levels can be written in terms of a virtual address space and disregard problems of paging. At the third level a message interpreter is introduced to handle communications between the console keyboard and higher level processes. The fourth level contains processes concerned with buffering input and output. The fifth and sixth levels introduce independent user processes and the operator.

One of the advantages of operating in this way, according to Dijkstra, is the ability to test the system at each abstract level before proceeding to the next level. This is done by forcing the system into all its "relevant" states at that level to verify that the system reacts according to the specifications. The difficulty of doing this lies in convincing onesself that no relevant state has been overlooked.

THE SODAS METHODOLOGY

The Parnas/Darringer methodology revolves around the programming language SODAS (for Structure Oriented Description And Simulation) which was conceived as a tool for the designers of computer systems. At each stage of the methodology a system is decomposed into a number of distinct component parts. This process is called <u>functional decomposition</u> and it defines a number of hierarchical levels of detail. The total system is on the highest level followed by primary subsystems, subsystems of primary subsystems, and so forth until one reaches the most detailed components at the final level.

The original SODAS language allowed only strictly nested components with explicit means of communication and was fairly closely restricted to the design of certain kinds of hardware modules. A new language called SOCS is being developed [Par 69] for the simulation and implementation of an Operating Computer System (OCS), where an OCS is defined as the hardware and software comprising a computer system. Much of this new work is based on ideas first introduced by Zurcher and Randell.

ITERATIVE MULTI-LEVEL MODELING

Iterative multi-level modeling is the name given by Zurcher and Randell [Zur 68] to still another methodology for structuring complex systems. The fundamental motivation for their methodology is to intermix the design, simulation, implementation, and evaluation of systems. The methodology is top-down in that the design of a subsystem first attacks the problems of what the subsystem does in terms of its interrelationship with other system components, before attacking the problem of how it does it. This approach allows the postponement of decisions regarding the realization of components until it is most appropriate.

The Zurcher/Randell approach has points in common with both the Dijkstra and Parnas/Darringer approaches. In particular there are two dimensions to the iterations being performed. First there are levels of abstraction which differ somewhat from Dijkstra's levels. The fundamental components of Dijkstra's levels are the software operations which constitute an abstract machine, while the Randell/Zurcher levels are based on an abstract machine of a given hardware configuration. For example, the Randell/Zurcher approach starts with a high level of abstraction regarding resources (i.e. unlimited processors or unlimited core storage) and proceeds to realistic assumptions (limited processors and limited core storage). Second, within each level of abstraction a minor series of iterations takes place of the functional decomposition type of Parnas/Darringer.

The basic Zurcher/Randell methodology can be visualized as a doubly nested iterative loop of the form:

```
DO UNTIL SYSTEM IS COMPLETE;
   ADD NEXT LEVEL OF ABSTRACTION;
   DO UNTIL LEVEL IS COMPLETE;
      DECOMPOSE CURRENT LEVEL OF ABSTRACTION;
      END;
   END;
```

Still another level of iteration is added by the fact that at each level of abstraction the system is simulated and evaluated. If the simulated performance is not satisfactory, the abstract level is modified in one of two ways. Either the system parameters are modified to achieve satisfactory simulated performance, or as a more drastic step, the structure of tho abstract level is modified resulting in another round of functional decomposition.

The jump from one level of abstraction to the next is taken when it is necessary to add new features to the design. Iteration by functional decomposition is done until each component is fully specified at each abstract level. The transition from one level of abstraction to another in the Zurcher/Randell model usually takes the form of adding constraints which limit the number of resources available. For example the first stage in an evolving design for a multiprocessor system [Ran 68] could use the abstraction of an infinite number of equivalent processors.

Randell has been experimenting with this design methodology using FORTRAN running under OS/360. When the design was felt to be complete he found that the simulation could not become the system because of its dependencies on FORTRAN, OS, and the IBM/360 [Par 69]. This experience reiterates the contention of Lang [Lan 70] that a systems implementation language should be independent of the operating system during its execution.

COMPARISON

In some sense the methodology proposed by Dijkstra and the ones proposed by Parnas/Darringer and Zurcher/Randell are incomparable. Dijkstra's emphasis is on implementation methodology while the emphasis of the other two projects is on

<u>design methodology</u>. Clearly Dijkstra had the whole operating system specified before he implemented THE in a bottom-up fashion. The design of his operating system, like most designs, probably started with general specifications and evolved into the more detailed specifications of the various levels.

On the other hand, the Zurcher/Randell methodology seems to be an effective modeling technique when used in connection with system design. Its effectiveness as an implementation methodology has yet to be proved on a real system. The methodology to be presented in the next several sections is meant to provide a greater integration of the design and implementation phases of the methodology. It is based on both the work of Dijkstra and Zurcher/Randell.

4.2 Basic components of a new methodology

We distinguish four distinct and largely independent phases in the design and implementation of an operating system:

(1) systems language implementation

(2) nucleus construction

(3) user job path description

(4) resource contention resolution

A systems language is an essential tool for rapid and concise solution of systems oriented problems. The nucleus consists of those routines written in the systems language which act as extended machine operations. User job path description, which is also called the path phase, is concerned with describing the gross structure of the operating system in terms of how user jobs traverse the system. In the resource contention resolution or resource phase, the more detailed design problems of resource definition and allocation are introduced. Most of the effort of this work will be concentrated on the last two phases, but first all four phases are described in more detail.

SYSTEMS LANGUAGE IMPLEMENTATION

Many of the basic structures and basic operations needed to construct operating systems are fundamental to all software systems. These should be the operations incorporated into a language for writing systems. Systems languages should free the programmer from tedious chores of coping with the hardware of the machine. In particular, the user of a systems language is not concerned with the mapping of higher level structures and operations onto machine structures and operations.

Advocates of higher level languages for systems are often subject to the criticism that systems written in this way are not as efficient as those written in assembly language. But such critics often ignore the tradeoff between execution efficiency and implementation efficiency.

There is some evidence that systems written in higher level languages may be more efficient than those written in assembly language because of success in specifying and modifying the global structure of programs. One of the earliest attempts to write an operating system in a higher level language was undertaken at M.I.T. in connection with the MULTICS project. The system was written in EPL (for Early PL/I) with only about 10% of the code being written in assembly language at the source code level.

Several interesting points were reported by Corbato [Cor 69] regarding PL/I as a tool for system programming. First, he claims that the number of lines of undebugged code produced per unit time is about equal for both assembly language and higher level languages. This means that a given algorithm can be implemented an order of magnitude faster in a systems language. Second, he found that on the average good EPL code was 2-3 times less efficient than "optimum hand code." Finally it was found that three major design changes could be implemented quickly (1 to 4 man months) at great savings in the amount of code generated (in one case the code was reduced from 50,000 to 10,000 words).

It should be remembered that PL/I was not specifically designed for implementing systems, nor was EPL a particularly good implementation of PL/I. However the MULTICS experience shows the feasibility of using a high level language for implementing systems and that by hand optimizing a relatively few sections of code, the efficiency of the system can be greatly enhanced.

NUCLEUS CONSTRUCTION

Once one has the systems implementation language with which to conceptualize and express problems, it is possible to build the special purpose tools to solve the particular problems of:

(1) process definition

(2) process interaction

(3) data definition and manipulation

(4) resource management

(5) input/output control

Primitives that solve these and other particular problems constitute the system nucleus. They should be chosen so as to be applicable to a wide variety of different modes of operation. The nucleus should be the intersection of all that is basic to many different types of operating systems. From a given nucleus one should be able to build a variety of batch, time sharing, or real time operating systems.

The concept of a nucleus is not new. Leonard and Goodroe [Len 64, Len 66] presented an early version of the idea in their work on extensible machines. Their basic notion was "one of extending the operation repertoire of a computer by means of a collection of programmed operations that effectively provide to the user an 'idealized' or higher level computer." They called this idealized computer a Basic Computer. Brinch Hansen [Bri 69, Bri 70] has built the RC4000 operating system from a nucleus of primitive operations.

Since it is the purpose of this chapter to present a methodology for the design and implementation of operating systems, we will not dwell on various alternative system nuclei. However it should be stated that the fundamental concepts in this area should be process definition, control, and interaction, and that the REQUEST and RELEASE primitives introduced in Chapter 2 provide a good start in this area. Clearly more research is needed to determine what constitutes a good nucleus for an operating system.

Another crucial point concerning the relationship of nucleus routines to the third and fourth phases of the methodology is the issue of simulation primitives. Particularly important are those primitives which coordinate real and simulated time. These operations were discussed in the last chapter.

PATH DESCRIPTION

After the basic building blocks are formulated, it is possible to start designing and constructing a particular operating system. It is at this point that we start working from the top-down rather than from the bottom-up (which was the case for establishing a systems language and the nucleus).

At the most abstract level of a total computer system we find the user. Thus the system must first be based on the user's expected behavior in terms of his input/output interface to the computer system. The basic entity at this level is a user job and the sole purpose of the system is to process user jobs. At this point of the discussion we limit ourselves to batch processing systems. In this case we have one (or more) streams of user jobs entering and leaving the computer system. (In time sharing systems the concepts of users and user jobs would be identical, but the concept of flow of user jobs would be modified).

Under these assumptions the basic purpose of a computer system is to process one or more streams of user jobs, so at the most abstract level we can view it as shown in Fig. 4.1. The simplest case would have one input stream and one output stream. The only interesting characteristic of a job from the user's standpoint at this primitive level is the elapsed time between entry on one of the input streams and exit from one of the output streams (the <u>turnaround time</u>). The turnaround time has two components, the time spent waiting to enter the computer system and the time spent in the computer system.

```
                        ---------------------
USER JOB 1 ----->|                   |
                 |                   |
INPUT    2 ----->|       BATCH       |
           .     |                   |-----> 1  USER JOB
STREAMS    .     |     COMPUTER      |
           .     |                   |-----> 2  OUTPUT
         n ----->|      SYSTEM       |
                 |                   |  .       STREAMS
                 |                   |  .
                 |                   |  .
                 |                   |-----> m
                 ---------------------
```

Figure 4.1 Global view of a batch computer system

If we consider the user job interarrival rate and the time in the system as independent random variables associated with each user job then we can model the system at this primitive level. The system will be characterized by 3 variables:

(1) the number of input streams

(2) the number of output streams

(3) the number of jobs that can be processed in parallel

The user job will be characterized by 4 variables:

(1) the number of the input stream

(2) the number of the output stream

(3) the interarrival time at the input stream

(4) the processing time within the computer system

The output of the model would include distributions (including maximum and average values) for

(1) the turnaround time

(2) the number of jobs in the system

The system is so simple at this abstract level that it is subject to analytical solution for a wide variety of different distributions for interarrival time and processing time. As increased complexity soon makes mathematical solution intractable, however, the analytical approach must be abandoned from the start.

The purpose of the foregoing discussion is to establish a starting point from which more detailed descriptions of the system are designed and implemented. In this simple case there are essentially only two states which characterize the

path of a user job through the system, namely its membership in one of the input streams, and its processing by the computer system.

The path phase of the proposed methodology is concerned with making successive refinements to this description of the path of a user job through the computer system, and simulating the system at each level. This procedure specifies the basic structure for the operating system and provides mechanisms for deriving data about the flows of user jobs through the system which are useful in the final implementation for purposes of evaluation.

As an example, the next level of refinement of our primitive computer system could be to divide the system into three subdivisions called respectively input service, main processing service, and output service. This is shown in Fig. 4.2. A further refinement would divide main processing into a loading phase and an execution phase.

During the path phase of the methodology the computer system can be conceived as a tree such that each <u>processing node</u> of the tree represents a stop along the path of a user job at a given level of abstraction. The descendents of a node represent any refinements to processing at a node. The path of a user job through the system is described by a left list traversal of the tree. A <u>left list traversal</u> is recursively defined by the algorithm:

(1) visit the root

(2) traverse each subtree of the root starting with

```
      ┌──────────────────────────────────────────────────────┐
      |                                                      |
      |                                                      |
      |              BATCH COMPUTER SYSTEM                   |
      |                                                      |
      |                                                      |
      |   ┌───────┐          ┌───────┐          ┌───────┐    |
 1 ---+-> |       |          |       |          |       |    |
      |   |       |          |       |          |       |--+--> 1
 2 ---+-> |       |          |       |          |       |    |
      |   |INPUT  |   ┌──┐   | MAIN  |   ┌──┐   |OUTPUT |    |
      |   |       |->|RS|->  |       |->|RS|->  |       |--+--> 2
      |   |PHASE  |   └──┘   |PHASE  |   └──┘   |PHASE  |    |
      |   |       |          |       |          |       |    |
 n ---+-> |       |          |       |          |       |    |
      |   |       |          |       |          |       |--+--> m
      |   └───────┘          └───────┘          └───────┘    |
      |                                                      |
      |                                                      |
      |            RS - RESOURCE SEMAPHORE                   |
      |                                                      |
      |                                                      |
      └──────────────────────────────────────────────────────┘
```

Figure 4.2 Refined batch computer system the leftmost subtree and continue to the rightmost subtree

Fig. 4.3 shows the three level tree described above with the nodes numbered in left list order.

In any tree representation the leaves represent the nodes at which most of the processing occurs. Those nodes which are not leaves are called <u>routing nodes</u> and serve to order the path of a user job through its subtrees and gather statistics on the flow of user jobs. Each node may consist of a number of identical processes which oversee the processing of a user job at a node. Each such process communicates with the system via two resource semaphores which act as a source and sink of user jobs processed at the node. Processing nodes get user jobs from a source by executing a REQUEST and relinquish user jobs to a sink by executing a RELEASE. These processes exist in the simulations during the path phase of the methodology and are retained in the final implementation.

Simulations of the system during the path phase of the methodology are useful for at least four reasons:

(1) The basic structure is established.

(2) Many derived statistics are retained in later versions of the system.

(3) Potential bottlenecks may be found.

(4) User job "resources" are set up for each phase.

By establishing the path of a user job through the system we have fixed a fundamental component of the logic of the computer system. That this must indeed be done has long been

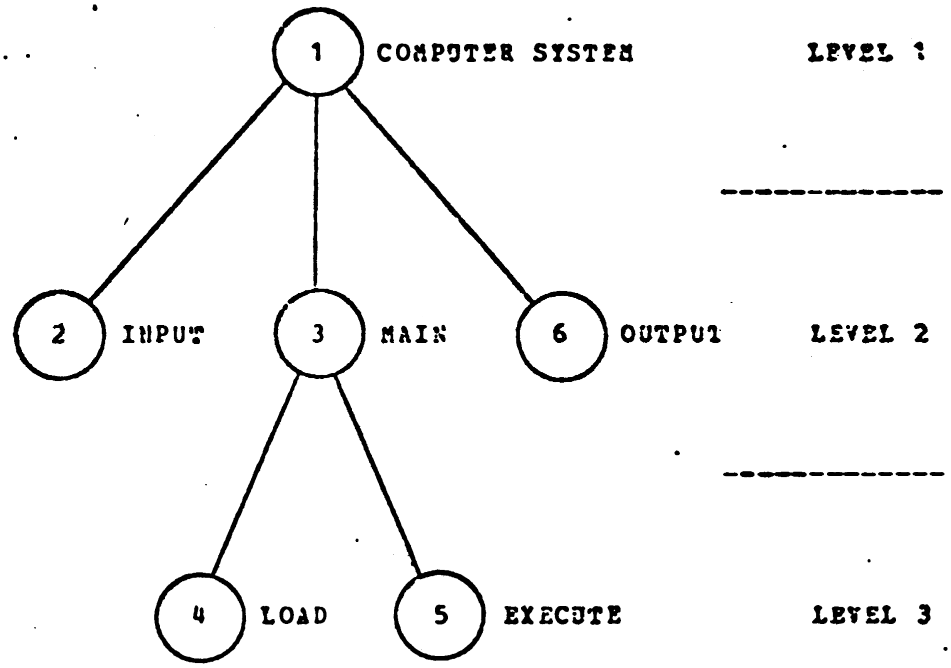


Figure 4.3 Tree representation of the path of a user job

recognized and the standard method for describing operating systems starts with a description of how user jobs move through the system.

The second reason for simulating is that many of the statistics derived at these early levels are the same statistics that are desired in the final implementation. Turnaround time is the obvious example of this. Since the simulation becomes the implementation these statistics gathering facilities are retained for later use.

Thirdly, these models can make the implementers aware of potential bottlenecks in the system by making a wide variety of hypothetical assumptions about how long a job spends in particular nodes along its path. This enables the designers to pinpoint critical nodes and guide them in future decisions about particular parts of the design.

The types of hypothetical questions that might be answered by this kind of analysis are the following: For a given distribution for interarrival time and node times, will all the queues remain bounded in the long run? Where do the largest queues build up? How can certain processing phases be speeded up by adding software or hardware?

Finally, since each processing node has a source and sink for user jobs we establish the resources for each of these nodes. Using the terminology of Chapter 2, each user job is a resource to a processing node in the path of a user job. If no user job is ready to be processed at a node, the job processor must go to sleep. If there are more user jobs than processes, the user jobs are queued on the inventory list of the resource and processed at a later time.

RESOURCE CONTENTION RESOLUTION

In the fourth phase of the evolutionary methodology resources are added in a series of steps so that the interactions among user jobs and systems processes due to their contention for limited resources is taken into account.

Recall the distinction between two kinds of resources. Those resources introduced as sources and sinks for user jobs between processing nodes are consumable resources and are not the object of competition among processes. Each process at a processing node is equivalent so it is irrelevant which waiting process is assigned to handle the user job. The resource is merely introduced to establish a communication link between two processing nodes.

The resources that are the subject of this subsection (serially reusable resources) are characterized by the fact that various processes request the resource which is in relatively short supply. "Short supply" means that there is a non-zero probability that when a process requests a resource of a particular type, the request cannot be granted. Examples of these resources are core storage, disk storage, channels, and peripheral devices.

The basic idea is to successively set up a number of resource semaphores representing serially reusable resources and then modify the processing nodes to take this resource into account. This is done by introducing the appropriate REQUESTs and RELEASES for these resources. The nature of these requests will either be introduced as parameters of the user jobs or as parameters of the system. Again the system is modeled as each set of new resources is added to the system.

At this stage of the methodology new processes may have to be added to the simulation. For example an internal process has to be introduced to simulate each peripheral device as it is added to the system. The processes at each of the processing nodes become more realistic and processing times become more a function of system interactions than a function of predetermined estimates.

As the resources are being introduced the data processing capabilities of many of the processes can be increased until they are in their final form. At this point the process ceases to be simulated except for its synchronization of real processing time with simulation time to allow the rest of the system to be simulated. Finally when all the resources have been introduced and all simulated hardware is replaced by actual hardware, the computation of simulated time can be dropped and the simulation becomes the implementation. The next several sections will give a more detailed description of user job characterizations, statistics gathering, and variability at each of the levels.

4.3 Characterizations of user jobs

At each simulation level the input (in addition to various system parameters) consists of a stream of user jobs. As the sequence of simulations become more detailed, so do the job characterizations.

In order to distinguish one job from the next it is necessary to establish conventions for job control. Although there is no reason to fix a permanent job control language for all simulation levels it may be more convenient to do so. Basically all one has to do at the most abstract levels is distinguish the beginning of a new job by some kind of job header. This serves to identify the job and later may establish various limits for processing time, lines printed, lines punched, etc. As the simulations become more detailed it may be necessary to introduce other kinds of control statements which would define data sets, invoke language or other processors, or mark the end of a job.

In the early stages of the simulation when the computer system describes the path of a user job through the system, the user job is an n-tuple where the i-th component represents the estimated time that the user job remains at processing node i. The process at the node then merely invokes a delay statement which puts the process in charge of the user job on the DELAY queue for the specified amount of time.

These values represent the elapsed time spent at a node rather than the CPU time required to process the user job.

Thus to be completely accurate in predicting these values it would be necessary to know what interactions take place between user and system processes at more detailed levels. As an example consider the elapsed time required to SPOOL a job from cards to an auxiliary storage device. This time consists of processing time, I/O time, and (possibly) time spent waiting for the channel, empty buffers of auxiliary storage. For this processing node, however, it would be reasonable to guess that the I/O time would be dominant. Since the I/O time is in turn directly proportional to the number of cards read, we can obtain a reasonable estimate of the elapsed time spent at this node. To estimate the effects of severe buffer, drum track, or channel contention these figures would be varied upward.

In the resource phase of the methodology the resources required by the user job are added. They can be expressed either directly or indirectly. Typical resources that would be needed by a user job are user core for execution of the user job; secondary storage space for the user job program input data, and output; peripheral devices; and the central processing unit.

These parameters may be expressed directly by including them as part of the user job n-tuple. For example we could specify that a user job requires 8 drum pages for its program and data, 6 core pages for execution, and 11 drum pages for output. However in some cases we may facilitate the evolution of the simulations by specifying these parameters indirectly through dummy program and data records so that the input service process can be developed to the point that actual cards are being read through buffers and then buffered onto the drum. This input service routine would then interpret the JCL and be in its final form except for its coordination with real time and communication with simulated peripherals.

When this happens the elapsed time estimates for the processing nodes must be replaced by estimates of processing time since this is one of the resources for which processes at all processing nodes are competing. The elapsed times of the various nodes become derived statistics for each user job rather than inputs. It is now possible to compare the original guesses with the derived results of the more detailed simulation to determine if and where bottlenecks have been introduced.

Program behavior also becomes important in this phase of the methodology. The performance of the system may depend critically on the average ratio of I/O time to computing time and the distribution of the I/O time over the lifetime of a user job execution. It is therefore important to be able to specify different user job behavior patterns at this level of simulation. The most accurate way of doing this is to specify the user job execution as a sequence of compute times separated by I/O calls. This expresses exactly the job's resource requirements in terms of processor time and I/O time, and also in terms of auxiliary storage if the job's output is being SPOOLed.

It is probably not convenient to express this sequence directly as part of the user job stream. Satisfactory results should be obtained by choosing one or several distributions for generating the sequence at execution time by producing pseudorandom numbers. The job's behavior can then easily be described by a distribution for the length of the time slices and one or more distribution parameters such as the mean and variance.

Fortunately, reasonable estimates of program behavior are available through tests which have been run on a wide variety of user jobs. Freibergs [Fre 68] presents one example of a study done on the dynamic behavior of programs.

Finally, in the limit, the job characterizes itself. The job itself represents all the demands (mostly implicitly) for resources of the computer system. The distribution of I/O operations are determined by the number and type of machine operations that it executes. Its storage requirements are determined by its program size, input, and output.

4.4 Statistics gathering capabilities

At each level, statistics are gathered on the performance of the system being simulated and on user jobs being processed. The statistics gathered become more detailed and specific as the simulations become more detailed. The primary goal of the statistics taken on the system is to provide feedback to the designer-implementer so that he may make appropriate changes to the existing system and make intelligent decisions to extend the system.

Another objective of the statistics concerns their use in the final implementation. The statistics gathering facilities of the simulation should remain to a certain extent in the final implementation as a software monitor of system performance. Such a software monitor can be used for fine tuning of the system, for detecting changes in usage patterns, and for pointing out the need for hardware modifications.

Most software monitors available today [Cos 70, Kol 71] are appendages to existing operating systems which were conceived only after managers realized that they had little idea of how well or how poorly their multiprogramming systems were performing. It is believed that a software monitor which is developed as an integral part of the design of an operating system will be both more efficient and more flexible.

One aspect of the proposed flexibility would be the ability to load a system with any desired level of statistics gathering capability. For example it is probably desirable to have at least three levels of monitoring. Low level monitoring would take place during most production hours so that the monitor would consume a minimum of system resources. Periodically (say every week or two) the system would be monitored at the intermediate level to get more detailed information on system performance and use. High level monitoring would be used infrequently and aperiodically to diagnose software or hardware problems, to collect data to use as the basis for major decisions regarding changes to software or hardware, and for research in the more detailed aspects of system behavior.

The problem with all software monitors is that they consume system resources just as any other user or systems processes do so that the measuring device biases the measurements. Thus there must be assurances that either this effect is small enough to be considered negligible, or that the software monitor corrects this bias. One software monitor written for an IBM 360, Model 91 takes 1.5% of user core and only .1% of the available CPU cycles [And 69].

During the path phase of the methodology the statistics describe the global behavior of user jobs, namely system turnaround time and processing node turnaround time. Also at this level we can collect a wide variety of statistics on the user job queues associated with each processing node, including average and maximum length and average and maximum waiting time.

Statistics gathered at this stage of the methodology are frequently refinements of statistics gathered on previous levels as each processing node is refined. For example, total turnaround time of a one level system could be refined into six components in a two level system: the elapsed input, main processing, and output times and the time spent waiting to enter each of these nodes.

In the fourth phase of the methodology we add resource utilization statistics as each of the resources is added. This is a relatively simple matter since all resources are handled uniformly through resource semaphores. Associated with each resource semaphore we need only keep track of the changes to the inventory list and waiting process list. Usually the data for the statistics are stored along with the resource semaphore data structure and are updated whenever there is a REQUEST or RELEASE. For low levels of systems monitoring, only certain of the more important resource semaphores are monitored in this way.

Statistics at any particular level of simulation are usually retained at subsequent levels. In fact some of the statistics introduced in early simulations are among the best indicators of overall system performance (turnaround time, throughput, number of jobs in the system). Thus in the completed implementation we have a number of levels of statistics reflecting different levels of detail and abstraction. Managers will be more interested in the grosser aspects of systems behavior (as it affects the user) while the systems programmers will be more interested in more detailed performance measures (as they affect the system).

Another measurement that is made in the monitoring of a system is the collection of statistics on user jobs. Installation managers can use user job statistics to help determine pricing strategies to discourage or encourage particular habits among users. If there are bottlenecks caused by a shortage of resources of a particular type such as drum storage, policy decisions which raise the price of drum storage relative to disk storage could be implemented.

In order for all these measurements to be useful in the design and implementation of the operating system it is necessary to have simulations which provide a wide degree of feedback at each level. The next section discusses the ways in which it is possible to modify the simulation at each level to obtain a diversity of statistics on system performance.

4.5 Variability at each level

As before we consider the systems language and nucleus fixed and deal only with the path and resource phases of the methodology. The ability to vary the simulation at each level is necessary for two reasons:

(1) to suggest changes for the existing design

(2) to guide future decisions in the evolving design.

First we deal with the path phase of the methodology.

THE PATH PHASE

Even at this relatively simple stage of the modeling we have a fairly wide latitude for making changes to the system and for learning about the system's final behavior. There are three areas in which we can vary the simulation:

(1) user jobs

(2) the number of processes at each processing node

(3) the schedulers for each processing node

User job variability at this phase of the methodology is primarily restricted to modifications of the time spent at each processing node. This may be done to reflect either a change in the job mix or a change in estimated system performance. For example consider the system with three processing nodes shown in Fig. 4.2. To reflect a job mix with more input or more output we would increase the amount of time spent in nodes 1 or 3 respectively. To reflect longer processing times the amount of time spent in processing node 2 would be increased.

The simulation of poorer system performance can be illustrated by the same example. The designer can answer the hypothetical question "What would happen if performance in processing node 1 is actually 50% worse than has been predicted for a given set of user jobs?" To answer this question, the processing time for node 1 would be increased by 50% for all user jobs.

Changes in the user job characterization at this phase suggest modifications to the existing system as well as direct future work on the system. If the experimenter systematically varies the time spent in processing node, he discovers which node or nodes have the largest effect on the total system behavior. These nodes are marked as critical nodes and should receive much of the effort as they are implemented on successive levels.

The number of processes at any processing node is another degree of variability at this level of simulation. The number of processes at a node may be software or hardware dependent. Again referring to Fig. 4.2 we find that nodes 1 and 3 are hardware dependent since there is only one input service process per card reader and one output service process per line printer. The number of processes in the main service node, however, is independent of hardware and is limited only by resources that these processes use.

Varying the number of input or output service processes would test the result of adding or deleting input and output streams. The number of main service processes, on the other hand, determines the maximum number of user jobs that may be processed in parallel. Increasing this number tests better processor service. Since jobs passing through a processing node in the path phase are 100% overlapped it would be unrealistic to increase this number indefinitely (except perhaps to determine the total computing power required to avoid bottlenecks at this processing node).

In a multiprogramming system it is desirable to execute several user jobs in parallel so that all the execution cycles of the processor are used. When one or more user jobs is blocked for I/O there should be another one which can continue execution. Of course the number of jobs that can execute in parallel is limited by the core storage constraint so that this number cannot be increased beyond a certain point.

The estimated time spent in the main service processing node is directly proportional to the number of processes serving user jobs in the main processing node. If a user job executed in the absence of other user jobs in time n, then the time taken to process m equivalent jobs in parallel is $n*k(m)$ where $k(m)$ is a function giving the constant of proportionality. The function k depends on the amount of interference between user jobs as they compete for system resources. If jobs are well balanced in their use of compute and I/O facilities it is possible that $k(m)$ will be significantly less than m.

The third area of variation is that of schedulers for processing nodes. These are the allocators for the resource semaphores that decide which user job is to be chosen next for any particular processing node. The simplest allocator would choose jobs in a FIFO ordering. Modifications of this might include shortest processing time schemes or priority schemes.

If it were found for example that a shortest processing time algorithm would significantly improve average turnaround time without degrading overall system performance, then this could be chosen as one alternative for incorporation into a more detailed version of the system. This decision would have two side effects. First it would imply a policy decision favoring short user jobs at the expense of longer ones, and it would introduce an extra expense in coding and execution time due to a more complex algorithm. These factors may or may not be obvious from the statistics gathered about the model on this level.

Many of the possible changes in the path phase suggest that emperical results could be useful as approximations to partial derivatives of some measure of the performance with respect to a change in some input parameter. Then, from a given system configuration, we can determine which of a number of changes has the most beneficial effect on overall system performance. Furthermore, if we assign a cost function to each of the independent parameters, it is possible to determine the changes which would produce the most improvement per unit cost. This area of optimizing system performance deserves additional study.

THE RESOURCE PHASE

During the resource phase of the methodology the results of the simulation become much more relevant to the final simulation. We distinguish 4 different kinds of variability during this phase:

(1) user jobs

(2) number of resources

(3) organization of resources

(4) allocators of resources

The variability of the user jobs at this phase was discussed extensively in the last section so we simply point out that the number of parameters that can be varied in a user job stream has increased tremendously. Particularly important is the modeling of dynamic program behavior.

The most sensitive independent parameter during the resource phase is the number of each type of serially reusable resources available to systems and user processes. Most critical is the amount of core storage available. Also, the amount of bulk and secondary storage is important for operating systems with virtual memories. Other examples of important resources are peripherals and communications channels.

Modifications at this level are often governed by financial constraints rather than by constraints on system behavior. Experimenters try to maximize system performance based on a particular budget. They test such things as the marginal utility of trading a core box for a drum. The idea is to find the optimal resource configuration for a given expenditure. Another consideration might be to strive for a configuration that could be most easily expanded to accommodate a heavier work load.

The concept of resource organization applies particularly to storage devices. The questions that should be answered at this level are ones such as "What is the best way to keep track of blocks of storage which are free or in use?" Alternatives are to use tables or linked lists stored in various levels of the storage hierarchy. In this stage of the development a file system might be introduced into the simulation as a sequence of hierarchical levels such as that proposed by Madnick [Mad 69] or Daley and Neumann [Dal 65]. The problems unique to simulating file systems have not been studied in any detail.

Closely related to the problem of organization of storages is the choice of an algorithm to allocate storage to a particular process. For example to allocate core storage according to a first fit strategy the free storage list could be ordered randomly. If a best fit strategy were chosen, the free list might be ordered in ascending order according to block size. Other modifications to the resource allocation algorithms could test the feasibility of adding more parameters to the allocation strategy, such as the process priority, the estimated length of use of the resource, etc.

The modifications made in all phases of the simulation and implementation come into four categories:

(1) modifications made when the system is generated

(2) modifications made when the system is loaded

(3) modifications to data

(4) modifications during execution

This distinction is important because the ease of modification varies greatly from case to case. In the simulation stages, modification should be coded as parameters of the simulation so that a wide variety of situations can be simulated. Recompilation of the system to make a change should be avoided. Modification during execution is a technique which can be carried over to the final implementation to reflect different strategies under different conditions. For example the free drum track allocator may have to be changed when there is a chance that the system might deadlock because of a lack of drum tracks.

4.6 Evaluation

One of the problems inherent in any methodology for computer system design is the choice of the next step from among many possible next steps. For this reason it is a desirable feature of a methodology to limit the number of possible next steps without unduly restricting the designer. We would like to reduce the number of possible false steps to zero.

The choice of the next step in this methodology can be guided by a number of basic principles. The first is that general, system wide decisions should be made before specific local decisions. This principle is reflected in the distinction between the path and resource phases of the methodology.

Another general principle for the resource stage of the methodology is called functional discrimination. That is, resources related in function should all be introduced at the same time. For example, the input buffer resource should be introduced at the same time as the card reader peripheral, which should be introduced at the same time as the JCL interpreter.

While there is no reason why one could not descend several levels on one branch of the implementation tree while staying at the same level on another branch, another general rule should be to maintain about the same level of detail throughout the system. This gives the simulation an overall balance which should be more amenable to critical analysis.

The ideas on design methodology presented here are relatively undeveloped and suggest many areas for future work. The contribution of this chapter is to make explicit one alternative for organizing the complexity of large operating systems. The approach described has been used successfully to design and implement the not-so-large operating system described in the next chapter. The applicability of the methodology to very large operating systems can only be proven by its adoption and use in such a system.

Experience gained using the methodology has shown that it achieves the objectives mentioned in the first section. Of course these are subjective opinions, but they are based on a qualitative comparison of about 10 implementations of the same operating system. These implementations provided a basis of comparison which would be impossible for very large systems.

Comparison with other existing or proposed methodologies is more difficult because it is difficult to define criteria of evaluation. One can only state the features of one methodology which are lacking in another.

The proposed methodology has many of the same advantages as the Zurcher/Randell methodology. Namely, the simulation at each level provides feedback to the designer about final system performance. The simulation also provides a basis for the documentation and a means of designing and evaluating new systems derived from an old system or extensions of an old system.

Because the proposed methodology is based on the Zurcher/Randell methodology its advantages over it are more subtle. Nevertheless it is believed that advantages exist in three areas. First, we have tailored this methodology around the assumption that a sequence of simulations would approach the implementation in the limit. The techniques for doing this were outlined in Chapter 3. In contrast Zurcher and Randell's approach was originally conceived as a modeling effort and they admit to grave problems in converting their simulation into an implementation.

Second, by simulating the path of user jobs through the system in its initial stages, the proposed methodology is believed to be more intuitive and natural than the Zurcher/Randell methodology. In other words, the methodology presented here is more closely related to what designers of operating systems have been doing all along. In contrast Zurcher and Randell seem to skip the path phase altogether.

Finally, the methodology's stress on process interaction and resource allocation as presented in Chapter 2 must be considered one of its strong points. Because of the importance of processes and process interaction in most operating systems, it is clear that this should be emphasized in the sequence of simulations leading to an implementation.

CHAPTER 5. A CASE STUDY OF A HYPOTHETICAL COMPUTER SYSTEM

To demonstrate the practicality and usefulness of the ideas appearing in the last three chapters, they should be put in the context of a real operating system. There are at least three alternatives for accomplishing this:

(1) Write an operating system for a real machine.

(2) Explain how the techniques could have been used in an existing operating system.

(3) Write an operating system for a hypothetical machine.

The first alternative was rejected because it would have required too much effort for a research project involving one person, and because of the lack of a readily available small machine. The possibility of describing the principles in terms of existing systems was rejected because of difficulty in explaining a new design methodology in terms of structures which might be inherently incompatible.

The third alternative was the one that was finally chosen, in part because of the availability of and experience with a hypothetical machine. The machine and its Multiprogramming Operating System (MOS) was designed by A.C. Shaw [Sha 68] for use as a term project in a course on operating systems at Cornell University. It has the advantage of manageable size while retaining many of the characteristics of real machines. A description of the machine and specifications for the MOS appear in Appendix I which is excerpted from a paper by Shaw and Weiderman [Sha 72].

Basically the hardware of the machine consists of a user storage and a supervisor storage, a drum storage, a card reader, and a line printer. Channels connect each of the peripheral devices with main storage. A CPU executes user programs in slave mode and interprets higher level language operating system code in master mode. Figures I.1 and I.2 show the virtual and real machines.

The operating system reads a stream of user jobs from the card reader and transfers them to the drum memory. Jobs are then loaded into the user store and executed with user job output being placed on the drum. Finally the completed job is transferred from the drum to the line printer.

The remainder of the chapter describes one implementation of the MOS in light of the principles and techniques presented in the thesis. The operating system was conceived, written, and simulated on four levels such that at the last level the only part being simulated is the hardware of the abstract machine.

5.1 The language and nucleus phases

The first stage of the methodology was completed when BCPL/360 [Kel 70], a version of BCPL [Ric 69], was chosen as the systems implementation language. BCPL was found to be a reasonably good choice because it has a simple syntax, but contains many good sequencing operations. It is quite natural to use, and at the same time it is close enough to machine language so that anyone familiar with its compilation will have a good idea of what machine code is generated by a sequence of BCPL statements.

The second stage of the methodology was to construct a system nucleus which would provide the basic operations for the operating system. The most important of these operations were the REQUEST and RELEASE semaphore operations of Chapter 2. Other primitives of the nucleus which have already been mentioned are CREATERS and DESTROYRS for creating and destroying resources and DELAY and SPEND which were primitives used to control the passing of elapsed and CPU time respectively.

Other routines in the system nucleus are concerned with data manipulation. These include routines to create and destroy queues and to enter and remove elements from queues. Some primitives are concerned with process creation and control. CREATEP and DESTROYP create and destroy a process respectively. The basic operation for process control is the RESUME primitive which enables the switch from the currently executing process to a specified new process. At a higher level of control a SCHEDULER primitive inspects the ready list and RESUMES one of the ready processes.

Other parts of the nucleus evolve as the operating system evolves. In particular, many of the allocators and queue handling routines associated with particular resource semaphores are added as those resource semaphores are added. The function of these routines is explained as the evolving operating system is described.

5.2 Level 1 of the MOS

In the MOS implementation the path stage of the methodology consists of two levels. The first level comprises three processing nodes called the input service node, the main service node, and the output service node. These nodes correspond to the three subsystems required to process a user job in its path through the system.

The simulation at this level consists of three resource semaphores and several processes, one each for the input and output processing nodes and a variable number for the main processing node. The first level is shown in Fig. 5.1.

The input service process is a cyclic process which performs the following sequence of actions:

(1) Read the next job characterization.

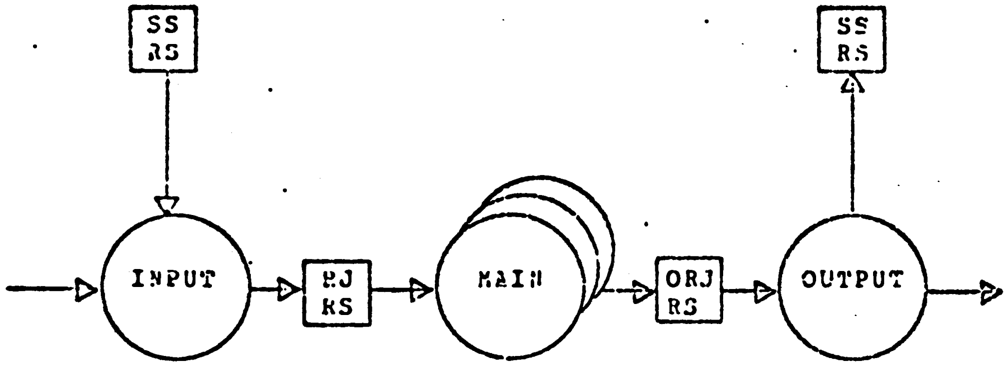
(2) Create a job control block (JCB) for the job.

(3) Delay for a specified amount of time to simulate the time required to SPOOL the job from a card reader to the drum.

(4) RELEASE the JCB to the Resident Job Resource Semaphore (RJRS).

(5) Repeat steps (1) to (4) for all jobs in the job stream.

If there are no more job characterizations to read, the batch is considered completed and the input service process destroys itself. (Under more realistic circumstances the process would go to sleep until more jobs appeared at the card reader). The



SSRS - Supervisor storage resource semaphore

RJRS - Resident job resource semaphore

ORJRS - Output ready job resource semaphore

Figure 5.1 Level 1 MOS design. Circles represent processes and boxes represent resource semaphores. Directed arrows represent flows of logical resources.

```
LET MAIN() BE
  $1 LET JCB = REQUEST(RJRS)
     REQUEST(DELAY,JCB.(MAINTIME))
     RELEASE(ORJRS,JCB)
  *1 REPEAT
```

Figure 5.2 Level 1 MAIN process

input service process requires supervisor storage space for the JCBs which it gets by making a REQUEST to the appropriate resource semaphore. The lack of supervisor storage is the only reason that this process may become blocked at this level.

The interface between the input service process and the main service processes is provided by the RJRS. The only processes which may wait for this semaphore are main processes and the only process to supply user job resources is the input service process. Therefore the purpose of the RJRS allocator is to match (possibly) distinguishable user jobs with indistinguishable processes. The simplest case is the FIFO allocator which treats the user jobs as if they were indistinguishable.

The processes of the main processing node at this level of abstraction consist of the three line BCPL program shown in Fig. 5.2. When the system is loaded each of the MAIN processes is placed on the ready list in a logically running status. First the process REQUESTs a resident job. If there are no resident jobs on the inventory list of the RJRS then the MAIN process goes to sleep until a job becomes available. Processing in the main processing node is simulated by a delay statement. This simulates the total time required for loading and execution of the user job.

When the processing for the user job at this node is complete the job control block is RELEASED to the Output Ready Job Resource Semaphore (ORJRS). The main processes each repeat these three operations and alternatively spend simulated time either on the process waiting list of the HJHS or the ready list.

The output service process representing the output processing node is another cyclic process which performs the following sequence of actions:

(1) REQUEST an output ready job.

(2) Delay for a specified amount of time to simulate the time required to SPOOL the completed job from the drum to the line printer.

(3) Destroy the JCB by releasing its storage to the free storage list.

(4) Repeat steps (1) to (3).

As with all the systems processes the output process starts on the ready list when the system is loaded. When it requests an output ready job it is put to sleep until some user job reaches this processing node.

STATISTICS

Two of the most important measures of system performance are evident at this level of abstraction. The most important measure from the user's point of view is the turnaround time (the elapsed time from recognition of the user job by the input service process until the job's last line is printed by the output service process). The amount of time the job spends in the hands of the operator is not measurable and is therefore ignored. The most important measure from the system's point of view is the throughput or the number of jobs processed per unit time. These two measures are usually highly correlated except perhaps in cases where the system is performing special services for privileged user jobs.

The turnaround time can be refined into five components, the time spent in each processing node and the time spent waiting in each of the user job queues. Also important at this level is the behavior of each of the user job queues which indicates the number of jobs resident on the drum. Based on the maximum queue lengths; the estimated size of user jobs including their program, input, and output; and the size of the drum, it is possible to determine order of magnitude estimates of whether the drum space is adequate to accommodate queues of the length found in the simulation.

Appendix II contains a sample of the output of the MOS for each of the four levels of simulation. For each run the output contains the new statistics of the current level plus the statistics gathered on all previous levels.

VARIABILITY

The two main variables to the simulation at the first level are the user job characterizations and the number of MAIN processes. The number of MAIN processes essentially reflects the designer's predictions about the number of user jobs that can effectively be processed in parallel by the system. One MAIN process indicates a the philosophy that no benefit is gained by multiprogramming.

If it was believed that the rate of flow of jobs through the MAIN service node could be increased by overlapping I/O and computing, then more MAIN processes would be appropriate. The number of MAIN processes must be restricted to the maximum number of jobs that can reasonably occupy main storage simultaneously. This number of user jobs represents an upper limit on the amount of possible overlap, but this maximum is rarely achieved.

User job characterizations consist of three integers at this level of abstraction. For the MOS implementation the estimates for input and output service were based on the number of cards read and the number of lines printed for each user job, and the real machine time specified for each reader and line printer peripheral operation. The final estimate was computed by a simple multiplication of the respective figures. This is an absolute minimum for the amount of time spent in each of these processing nodes and assumes that there is no channel interference, CPU time is zero, and time waiting for input or output buffers is also zero.

The time spent in the MAIN service processing node was derived from the number of pages in the user job program, and the number and type of instructions executed by the user job during its execution. The estimate was computed as the product of the number of processes in the MAIN processing node and the sum of the I/O time and CPU time required for loading and execution.

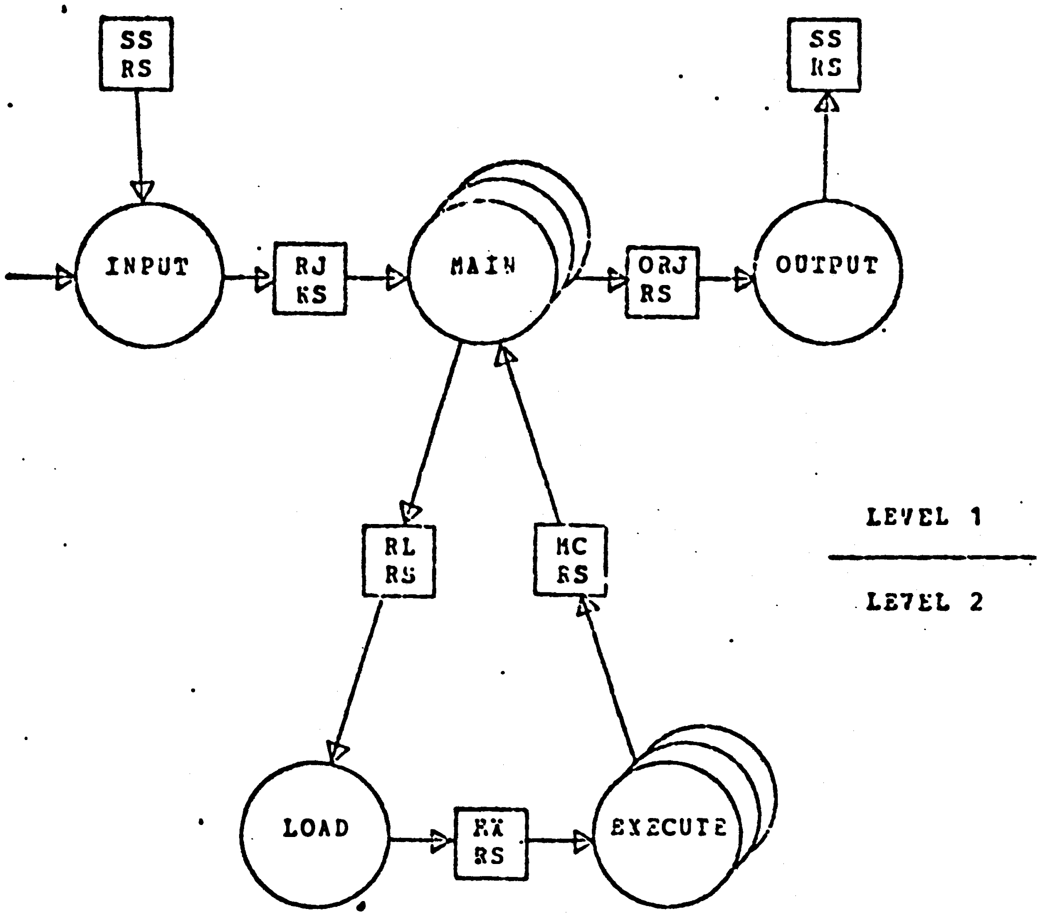
At all simulated levels statistics are obtained directly from the user jobs either by inspection (for number of pages) or by running the job on an existing system (for the execution time). From this exact representation of the user job we derive a characterization, consisting of processing node times, which is based on guesses about anticipated system behavior. These guesses are based on assumptions about waiting times in the various processing nodes.

5.3 Level 2 of the MOS

The second level of the methodology in the path phase introduces a refinement to the MAIN processing node. Fig. 5.3 shows the two level system with the descendents of the MAIN service node being a loading phase and an execution phase. One resource semaphore (EXRS) is introduced to pass user jobs from the loading phase to the execution phase and two resource semaphores (RLRS and MCRS) are required for communication between the first and second levels.

Clearly, the two levels could be collapsed into one by eliminating the main service subsystem and replacing it with the load and execute subsystems. The reason for keeping two distinct levels (at a small cost in time and space) is to maintain the simple structure of level 1 while adding new processing nodes. If the system were collapsed because of design changes, the software grouping associated with level 1 and its associated statistics would be lost, and would exist only in the system documentation (if at all).

This simple change to a new level illustrates a technique used whenever a process is refined. Namely, the spend or delay synchronization statements are replaced by more function specific statements. This phenomenon is shown by the BCPL code for the MAIN and EXECUTE processes in Fig. 5.4. The MAIN process is modified so that the delay statement of the one level system is replaced by two statements which communicate with the level 2 processes. Since we are still in the path



RLRS - Ready load resource semaphore

MCRS - Main complete resource semaphore

EXRS - Execute resource semaphore

Figure 5.3 Two level MOS design

```
LET MAIN() BE
  $1 LET JCB = REQUEST(RJRS)
     RELEASE(RLRS,JCB)
     JCB = REQUEST(MCRS)
     RELEASE(ORJRS,JCB)
  #1 REPEAT
```

```
LET EXECUTE() BE
  $1 LET JCB = REQUEST(EXRS)
     REQUEST(DELAY,JCB.(EXECUTETIME))
     RELEASE(MCRS,JCB)
  #1 REPEAT
```

Figure 5.8 Modified MAIN process and EXECUTE process of the two level MOS simulation phase of the methodology the RELEASE operation passes a user job to the loader while the REQUEST gets the same job back when the main service phase is completed.

In the two level system the user job characterization no longer includes a specification of how much time is spent in the main subsystem. Instead, the characterization contains estimates of time spent in the two descendent processing nodes. The estimate of load time is derived from the number of pages in the user job program, while the execution time is derived from the number and type of operations executed by the user job.

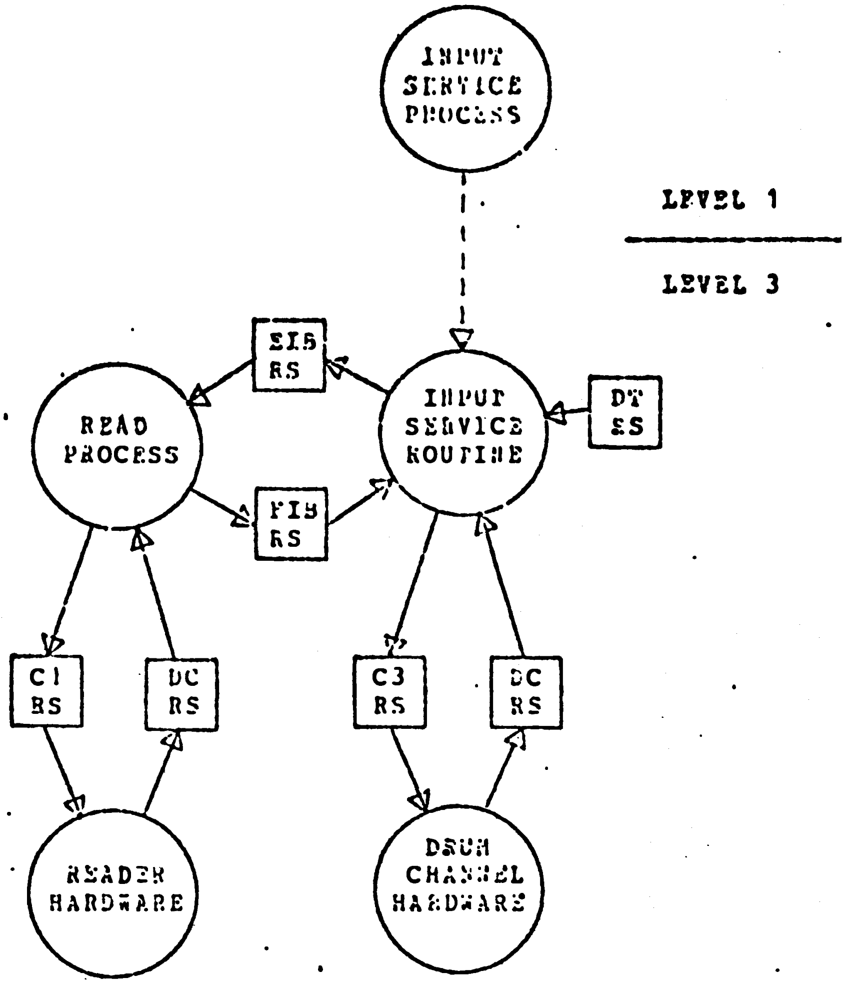
Additional statistics are derived from the two new resource semaphores. The main component of the turnaround time is refined into four components - time spent in each of the two new processing nodes and the time spent waiting to enter each of the nodes. Also of interest is the behavior of the new resource semaphores. Since the number of user jobs that can be in level 2 simultaneously is limited by the number of main processes at level 1, we see that the user job queues are bounded in size. This also means that there is no point in having more execution processes than main processes. Even though these queues vary within strict limits they provide indications whether either of these subsystems is causing a bottleneck in the system, depending on whether the average queue size is near zero.

5.4 Level 3 of the MOS

The third level of the system starts the resource phase of the methodology. It is primarily devoted to introducing (simulated) hardware and to filling in the black boxes left on levels 1 and 2. First, buffered input and output are introduced by setting up resources of empty and full input and output buffers. A reader subsystem is added which cyclically requests empty input buffers and which releases empty output buffers. A writer subsystem cyclically requests full output buffers, prints the contents of the buffer using the simulated hardware printer, and then releases the empty output buffer.

The delay in the input service processing node is replaced by a call to a more primitive routine which requests full buffers, requests a free drum track, transfers the information in the buffer to the drum using drum I/O hardware, and then releases the buffer as an empty input buffer. Similarly the delay in the output service node is replaced by a call to a routine which transforms empty buffers into full buffers by transferring the program and output from the drum. This routine also releases drum tracks. The refined input and output service nodes are shown in Figs. 5.5 and 5.6.

To accomplish the transition to level 3 we have added five new processes and ten new resource semaphores. Three of the processes and four of the resource semaphores are introduced to simulate the hardware peripheral devices - the card reader, the line printer, and the drum - as indicated in



EIBRS - Empty input buffer resource semaphore

FIBRS - Full input buffer resource semaphore

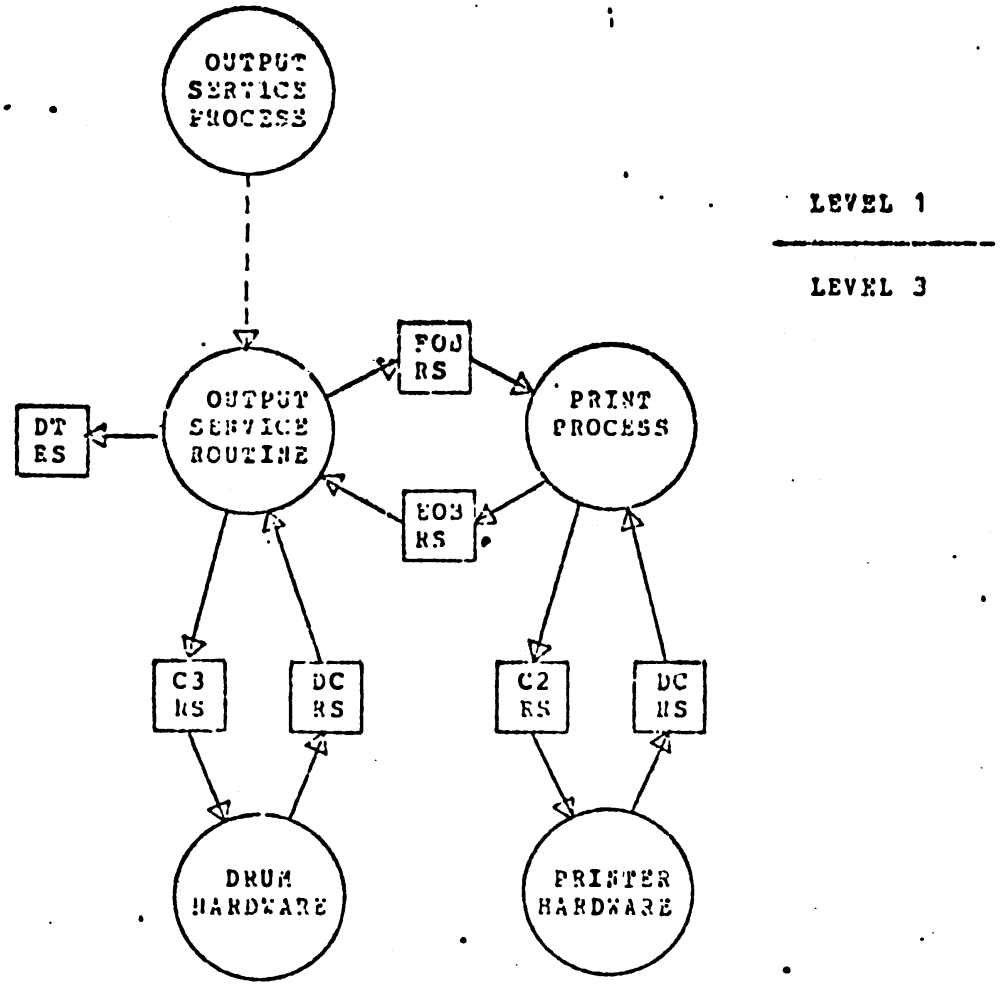
C1RS - Channel 1 resource semaphore

C3RS - Channel 3 resource semaphore

DCRS - Device complete resource semaphore

DTRS - Drum track resource semaphore

Figure 5.5 Three level input process



FOBRS - Full output buffer resource semaphore

EOBRS - Empty output buffer resource semaphore

C2RS - Channel 2 resource semaphore

C3RS - Channel 3 resource semaphore

DCRS - Device complete resource semaphore

DTRS - Drum track resource semaphore

Figure 5.6 Three level output process

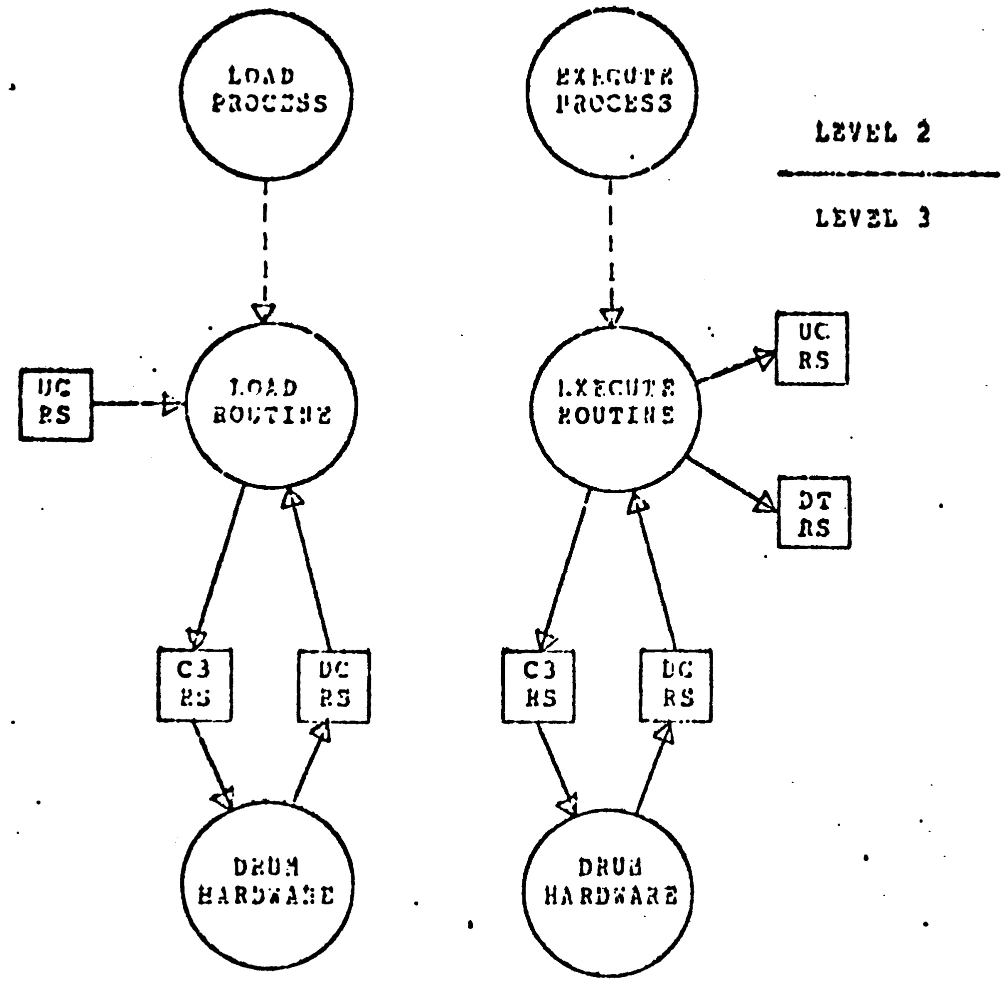
Section 3.4. The other two processes represent the software required to read cards from the card reader and print lines on the line printer.

Four of the resources are of the consumable type and transmit empty and full input and output buffers. A drum track resource semaphore is introduced to keep track of free drum tracks which are used by input service when jobs are SPOOLed to the drum and returned by output service as the job is SPOOLed from the drum to the line printer. A user core resource keeps track of user core blocks.

The refined input service process interprets the job control language, and sets up program and data files for the user job. The contents of cards other than control cards are irrelevent to the operation of the routine. The output service routine merely transfers the job's program and output to output buffers for printing.

The refined load and execute processes are shown in Figs. 5.7 and 5.8. No new processes are introduced by these refinements. Only the user core storage resource semaphore has not appeared before. The load routine consumes blocks of storage for the program of the user job and the storage is returned to the system by the execution routine when the execution of the user job has been terminated.

The loader routine first requests core for the user job's page table. It then opens the job's program file and loads the program page by page, updating a page table entry each time a page is loaded. The data transfer from the drum to



UCRS - User core resource semaphore

DTRS - Drum track resource semaphore

C3RS - Channel 3 resource semaphore

DCRS - Device complete resource semaphore

<u>Figure 5.7</u>
Three level
load process

<u>Figure 5.8</u>
Three level
execute process

user core is accomplished in the usual manner by sending commands to the drum channel and waiting for a completion signal.

The execution routine supervises the execution of a user job. Input and output files are opened so that the user job can read data from and write data to the drum. When the user job does an I/O operation the system intervenes to perform the operation for the user job by sending commands to the drum channel and then waiting for the completion signal.

USER JOBS

The user job stream in the three level system consists of user jobs with dummy programs and data. Examples of a dummy user job and the associated real user job appear in Fig. 5.9 and 5.10. The basic idea is that the number of program cards implicitly defines the amount of user store required and the number of input cards defines the amount of data associated with the user job. In this way all the data independent processing nodes (except the execution node) can be written in their final implementation form.

The execution of a user program is simulated as a sequence of time slices separated by I/O calls which read from the dummy data file and print on the dummy output file. This can be done in a variety of ways to simulate many different kinds of user job behavior as explained in Section 4.3. In the MOS implementation the I/O operations were placed at equal intervals in a simulated program based on the total number of

```
$MJBTRW110000010
     PROGRAM CARD 1
     PROGRAM CARD 2
     PROGRAM CARD 3
     PROGRAM CARD 4
     PROGRAM CARD 5
     PROGRAM CARD 6
     PROGRAM CARD 7
     PROGRAM CARD 8
     PROGRAM CARD 9
     PROGRAM CARD 10
$DTA
     DATA CARD 1
     DATA CARD 2
$FINTRW1
```

Figure 5.9 Dummy user job

```
$MJBTRW110000010
GD80LR82SR84LR80SR85LR08CR08BT57BT09SR86
CR90BT23LR84SR85LR17CR17BT31BT18SR84LR86
SR85LR08BT31LR79SR83LR98SR85LR99SR86PD80
BT00SR56LR90CR85LR91BT56CR85LR92BT56CR85
LR93BT56CR85LR94BT56CR85LR95BT56CR85LR96
BT56CR85LR97BT56LR98CR90BTXXSR56LR85CR90
BT54CR91BT52CR92BT49CR93BT46CR94BT43CR95
BT40CR96BT37CR85BT34                   =

  0   1   2   3   4   5   6   7  MOD 8
$DTA
  1   +   2
  2   +   7
$FINTRW1
```

Figure 5.10 Corresponding real user job
The author is indebted to T.R. Wilcox
for this program which adds two numbers
modulo 8.

I/O and non-I/O operations of a real job.

VARIABILITY

The three level simulation has a great latitude for variability. On the hardware side we can vary the speeds of any of the three simulated peripheral devices simply by changing a system parameter which governs the amount of time their corresponding processes are delayed when a command is accepted. The size of the user store or the drum store are easily modified by changing the initial inventories of the corresponding resource semaphores.

On the software side one of the most interesting possible modifications concerns the allocator associated with the drum channel. It should be noticed that the drum is the only channel for which there is competition among processes. Namely, the input service, output service, loading, and execution processes all require data transfers between core and drum. Thus there is a scheduling strategy implicit in choosing which command for the drum peripheral should be serviced next. For the MOS it was decided to assign each type of process a static drum channel priority so that channel time could be granted by the allocator to the highest priority process requesting it.

Another parameter at this stage is the number of input and output buffers available to the input service and output service processes. Again these modifications take place when the system is loaded by initializing the empty buffer resource semaphore inventory lists with the specified number of buffers.

Statistics gathered in the three level system are mainly of the resource utilization type. Average and maximum utilization statistics are realized for drum and user storage while average utilization figures are obtained for the data channels, the CPU, and the input and output buffers.

Also at this level, statistics are gathered for the dummy user jobs including the number of core pages, the number of cards read, the number of lines printed as well as the total run time based on the number of I/O and non-I/O operations specified for the dummy job. At the next level these statistics for user jobs are based on actual user jobs.

CHANGES TO PREVIOUS LEVELS

The changes to code on previous levels which are required to incorporate level 3 include changing three delay statements and a spend statement to calls to more primitive routines. The delay statements appeared in the input service and output service processes at level 1 and the load process of level 2 and the SPEND statement appeared in the execute process. The modified load process and the level 3 load routine are shown in Figs. 5.11 and 5.12.

Another change that is necessary at higher levels is the change to the job control block. It must be modified to incorporate such information as the location and size of user job files, card and line limits, and completion information.

```
LET LOADPROCESS() BE

  $1 LET JCB = REQUEST(RLRS)  || REQUEST READY TO LOAD JOB

     LOADROUTINE(JCB)         || LOAD THE JOB

     RELEASE(EXRS,JCB)        || RELEASE READY TO EXECUTE JOB

  #1 REPEAT                   || REPEAT FOR ALL JOBS
```

Figure 5.11 Modified load process

```
LET LOADROUTINE(JCB) BE
                              || DEFINE TEMPORARY VECTOR
  $1 LET COMMAND = VEC 5
                              || DRUMTRACK IS DRUM ADDRESS
     LET DRUMTRACK = JOB.(FIRSTPROGRAMCARD)
                              || ALLOCATE PAGE TABLE
     JOB.(PAGE_TABLE) := REQUEST(UCRS)

     JOB.PAGES := 0           || LOAD PAGES UNTIL END OF
                              ||    PROGRAM OR MEMORY OVERFLOW
     WHILE NOTEND(JOB) & JOB.(PAGES) < 10 DO
                              || SET UP CHANNEL COMMAND
       $2 COMMAND.(TO@) := REQUEST(UCRS)

          COMMAND.(FROM@) := DRUMTRACK
                              || QUEUE COMMAND FOR CHANNEL 3
          RELEASE(C3RS,COMMAND)  || I/O PROCESSING

          REQUEST(DCRS)       || WAIT FOR COMPLETION

          JOB.(PAGES) := JOB.(PAGES) + 1
                              || UPDATE THE PAGE TABLE
          ENTER(COMMAND.(TO@),JOB.(PAGE_TABLE))
                              || TRACKS ARE CHAINED
          DRUMTRACK := DRUMTRACK.(NEXT)

       #2
                              || ERROR - EMPTY PROGRAM
     IF JOB.(PAGES) = 0 DO JOB.(COMPLETIONCODE) := 6
                              || ERROR - TOO MANY PAGES
     IF NOTEND(JOB) DO JOB.(COMPLETIONCODE) := 7

  #1
```

Figure 5.12 Load routine

Fortunately these changes are localized and do not require any modification to the resource semaphores whose resource elements are user jobs. The allocators for these resource semaphores need be changed only if the allocation strategy is based on new criteria at level 3.

5.5 Level 4 of the MOS

At level 4 the CPU hardware is introduced to permit the execution of real user jobs. This is done by modifying the execution routine introduced at level 3. With the CPU come its registers (instruction counter, accumulator, toggle, page table register, and timer) and four types of interrupts (protection, supervisor, I/O, and timer).

The routine is written so that when the execution process gets control, the CPU registers are loaded from the JCB and the CPU starts slave mode operation by transferring control to a microprogram in the read only memory. The microprogram fetches and executes instructions indirectly through the page table. An invalid address causes the protection interrupt flag to be raised. The instruction is then executed with get, put, and halt machine operations causing supervisor interrupts. An invalid operation code causes a program interrupt.

At the end of the instruction cycle the CPU time is simulated by incrementing the clock (by a SPEND primitive). If there are any interrupts pending at this time, there is a switch to master mode and control is transferred to the proper interrupt handling routine. When an I/O operation has been executed by a user program the associated interrupt handler executes a RELEASE to send the appropriate command to the channel scheduler. The execution process then waits for the completion of this operation by executing a REQUEST operation.

The processing of an I/O interrupt causes the execution of a RELEASE primitive which wakes the waiting process.

The timer, protection, and halt supervisor interrupts cause the termination of the user job execution phase. The job's core pages are released, the unread data pages are purged from the drum and finally the job is returned to the MAIN subsystem at level 1.

VARIABILITY

The only major new source of variability introduced by the four level system is in the area of program behavior. Since we are dealing with real user jobs, each different program can exhibit a different behavior. In the case of the MOS a batch of 57 user jobs was used to test the system. In another series of tests a number of batches of user jobs were generated automatically to test their effect on system performance. The results of these experiments are discussed in Section 5.7.

Other modifications that are possible at level 4 are changes to the CPU hardware to modify the amount of time required to execute a user job instruction. This would simulate different CPU speeds. Another change would involve the CPU timer. The MOS specifications call for it to be decremented every 10 CPU cycles. Changing this constant could mainly affect the accuracy of the timing.

One software change that could be introduced into the four level system is time slicing. Without time slicing it is possible for a user job in a loop to effectively prevent multiprogramming by dominating the CPU. If the user job were forced to give up the CPU every so often to allow other processes in the system a chance to execute, this bottleneck would be avoided.

STATISTICS

New statistics introduced into the four level system concern the interrupt handlers. In particular we can gather frequency counts on the four classes of interrupts or break this down even further by subclasses of interrupts. For example I/O interrupts can be broken down by channel. We can also make some measurements of the amount of work being done by the system relative to the amount of work done by user jobs. The measure chosen for the MOS was the number of process switches occurring per user job instruction executed.

CHANGES TO PREVIOUS LEVELS

The changes required to the three level system to accommodate the fourth level are rather insignificant due to the completeness of the three level system. The only processing node that is affected at all is the execution node. The new routine described above with the incorporated CPU and interrupt routines replaces the routine which formerly simulated user job behavior. The input, loading, and output processing nodes remain unchanged because they were independent of the particular user job programs and data which

[illegible]

[illegible] made to the simulated [illegible] to accommodate the introduction of [illegible]

[illegible] AND)

[illegible] explained in Section 3.4, the [illegible] raise the I/O interrupt [illegible] appropriate [illegible] register. Then, at the [illegible] cycle, the associated interrupt [illegible] the process waiting for [illegible] of the I/O operation.

[illegible] summarizes the [illegible] level MOS implementation [illegible] process [illegible], and resource semaphores.

```
              PROCESSES         ROUTINES       RESOURCES

LEVEL 1:   INPUT----------------┐           SUPERVISOR STORE
           MAIN----┐            |           RESIDENT JOBS
           OUTPUT┐ |            |           OUTPUT READY JOBS
                 | |            |
                 | |            |
                 | |            |
LEVEL 2:         | ├>LOAD-----┐ |           LOAD READY JOBS
                 | └>EXECUTE┐ | |           EXECUTE READY JOBS
                 |          | | |           MAIN COMPLETE JOBS
                 └--------┐ | | |
                          | | | |
                          | | | |
LEVEL 3:   READER         | | | └>INPUT     USER CORE BLOCKS
           PRINTER        | | └-->LOAD      DRUM TRACKS
           CH1HARDWARE    | ├---->EXECUTE   EMPTY INPUT BUFFERS
           CH2HARDWARE    └-┼---->OUTPUT    FULL INPUT BUFFERS
           CH3HARDWARE      |               FULL OUTPUT BUFFERS
                            |               CHANNEL 1 COMMANDS
                            |               CHANNEL 2 COMMANDS
                            |               CHANNEL 3 COMMANDS
                            |               DEVICE COMPLETE
                            |
                            |
                            |
LEVEL 4:                    └---->EXECUTE
```

<u>Figure 5.13</u> Summary of the processes, routines, and resource semaphores of the four level MOS implementation

5.6 Level independent considerations

SYSTEM GENERATION

In order to process jobs at any of the four levels of the simulation it is necessary to set up an environment for the simulation. This will be called an Initial Program Load (IPL) which is always done before the system can be started.

The IPL procedure consists of 7 steps:

(1) Establish a ready list and include the IPL procedure as the highest priority process so it can use the REQUEST and RELEASE primitives.

(2) Set up a resource semaphore for the system core space.

(3) Read in the system parameters for the level being simulated.

(4) Initialize global variables for the levels.

(5) Create processes for the levels by requesting system core for the PCBs and inserting them on the ready list.

(6) Create resource semaphores for the levels by requesting system core for them and initializing them.

(7) Destroy the IPL process and resume highest priority process on the ready list.

Steps (2)-(6) represent the IPL process. The crucial steps are (5) and (6) in which the environment for the operating system is established. In the path phase levels the processes associated with each processing node are created as are the resource semaphores which transmit user jobs. During the resource phase of the methodology, several more software and hardware processes are added as well as many resource semaphores which are initialized with the total number of resources of that type (usually supplied as a parameter). When the environment is finally set up, the IPL process disappears and the operation of the system begins.

SUPERVISOR MODE TIMING

The original specifications for the MOS stated that the abstract machine should be simulated as running in zero time in master mode. This meant that only user job instructions would consume time and the system would have no overhead. This is of course not a realistic approximation to real systems. The problem of introducing non-zero master mode execution times in the MOS lies in making a realistic correspondence between higher level language statement execution times and user job statement execution times. Depending on the ratio chosen the system overhead can be made as large or as small as desired.

A realistic compromise for this particular project would be to assign certain CPU execution times to certain software functions (rather than to each higher level language statement). For example, charge x units of CPU time to process each input card during input service, y units of CPU time to load each user job page, and z units of CPU time to process each interrupt. This would add somewhat to the realism of the project, but would not really say much about the overhead imposed by the operating system because of the inherent incomparability of systems programs and user programs in this system. Since system time would be based on specific functions rather than the algorithm used to accomplish that function, the system overhead would be invariant over all possible implementations provided the functions remain unchanged.

RECAPITULATION

It is appropriate to emphasize what has been accomplished by the four level simulation of the MOS. First we have developed the system in an evolutionary manner such that meaningful simulations exist at each level. Also this has been accomplished with a minimum of backtracking. That is, very few changes have been made to previous levels when any given level is added. Finally, and most importantly, in the four level system all the operating system software is in its final implementation state. Only the hardware of the abstract machine (the peripherals, the CPU, and the storages) are still being simulated. If the abstract machine existed we would have an operating system for it at this point.

5.7 Experimental results

Two series of tests were run on the MOS implementation described in this chapter. In the first series of tests 4 batches of jobs were generated automatically to compare the behavior of four different user job streams. Each batch was simulated at each of the four levels to determine the validity of the simulation at the different levels of detail. In the second series of tests the user job stream was a batch of 57 student written jobs. These jobs were run only on the full four level system to test the results of making various software and hardware changes.

In both series of tests user job characterizations were derived *a priori* before any simulations were performed. In the path phase of the methodology processing node times were derived as follows:

Input time = (Cards input) * (Channel 1 speed)

Output time = (Lines output) * (Channel 2 speed)

Load time = (Program pages) * (Channel 3 speed)

CT = (Total instructions executed) * (CPU speed)

Execute time = CT * (Number of execute processes)

Main time = (Load time + CT) * (Number of main processes)

SERIES I

User job streams in this series of tests were generated automatically by another program. This permitted the rapid formation of a number of batches of uniform and easily characterizable user jobs. The batches were generated with different compute-I/O ratios. The ratios chosen were 8, 4, 2, and 1 to 1. The total average execution time was kept the same for all the batches so that as the number of execution operations decreased, the number of I/O operations increased by an equal amount. A side effect of increasing the number of I/O operations is to increase the number of cards read or lines printed, or both. Hence increasing the compute-I/O ratio also decreases the amount of time spent in the input and output nodes. The ratio of input to output was 1 to 1 for all batches except the last for which it was 1 to 2.

The results of this series of tests is shown in Table 5.1. The accuracy of the simulations at the low levels of detail is reflected by the figures for throughput measured as the number of jobs processed per 1000 machine time units. If all the simulations were completely accurate these figures would be equal for a given batch of jobs over all simulated levels. The variation reflects the fact that the guesses for processing node times were not completely accurate. It is significant that the guesses, which were made without knowledge of process interactions occurring at more detailed levels, were sufficiently accurate to produce throughput figures which do not vary more than 10% for any batch and whose average maximum variation for the four batches is only 7%. The consistently higher throughput figures for level 2 over level 1 are due to a small overestimation of MAIN processing time at the first level.

Table 5.1 Series I test results

		BATCH I	BATCH II	BATCH III	BATCH IV
COMPUTE:I/O RATIO		8	4	2	1
THROUGHPUT:	LEVEL 1	10.0	9.8	9.7	8.3
	LEVEL 2	10.7	10.5	10.4	8.5
	LEVEL 3	10.6	10.5	10.0	8.5
	LEVEL 4	10.8	10.8	10.3	8.4
UTILIZATION: CHANNEL 1	LEVEL 3	.42	.53	.70	.58
	LEVEL 4	.43	.55	.71	.59
CHANNEL 2	LEVEL 3	.38	.50	.67	.94
	LEVEL 4	.39	.52	.68	.94
CHANNEL 3	LEVEL 3	.37	.53	.76	.89
	LEVEL 4	.38	.55	.77	.89
CPU	LEVEL 3	.95	.94	.91	.76
	LEVEL 4	.96	.97	.92	.76
MAX. RESIDENT JOBS:	LEVEL 1	29	22	14	14
	LEVEL 2	27	20	11	11
	LEVEL 3	16	9	5	3
	LEVEL 4	14	9	5	2
MAX. OUTPUT READY:	LEVEL 1	0	0	0	10
	LEVEL 2	0	0	0	8
	LEVEL 3	0	0	1	3
	LEVEL 4	0	0	1	3

Also significant in the throughput figures is that the variation from batch to batch at the lowest level of detail is roughly equivalent to the variation at the highest level of detail. Both level 1 and level 4 show that by far the largest variation in throughput between consecutive batches comes between batches III and IV. Using batch I as a base, throughput decreases by 2%, 3%, and 17% at the lowest level of detail, and by 0%, 5%, and 22% at the highest level of detail. These figures indicate that the simulations at the lower levels of detail are useful in predicting the same qualitative changes that occur in the final implementation, assuming reasonable guesses for processing node times.

Resource utilization statistics can only be compared between levels 3 and 4. The figures for the utilization of the three channels and the CPU show a maximum variation between the two levels for all batches of 4% and average variation of less than 2%. These figures indicate that the system with simulated user job behavior is an excellent predictor of resource utilization. Since there are no delay statements in the three level system, the only deviation from reality is in the behavior of user jobs. Therefore we would expect the simulation to be very accurate at this level.

Also shown in Table 5.1 are the maximum lengths for the user job queues in the resource semaphores for the main service and output service processing nodes. These are very sensitive indicators because they may in fact be unbounded for arbitrarily large batches of jobs. This, for example, would be the case when the flow rate of jobs through the input service node is greater than the flow rate of user jobs through the main service node.

The maximum queue size statistics are not as accurate as the throughput figures. In almost all cases the maximum figures for the lower level simulations are greater than the corresponding figures for the more detailed simulations. This is due to the fact that the time in the processing nodes has been underestimated and as a result flow rates are increased and queues are longer than at the final level.

However these statistics reflect the same qualitative changes for each of the four user job batches. At each simulation level the maximum resident job queue gets smaller as the system becomes more I/O bound. The output ready queue is nearly always zero except in batch IV where the increase in maximum queue size reflects the increased I/O activity.

SERIES II

The second series of experiments were designed to test the effects of changing various software and hardware parameters on system performance. The user jobs for these tests were written by several dozen people over a period of two years. Table 5.2 summarizes five characteristics of the batch of 57 jobs. In general they are shorter and have a rather high compute-I/O ratio (6 to 1).

A summary of these results is shown in Table 5.3. The first five test runs were devoted to finding a good strategy

__Table 5.2__ Statistics on 57 student jobs

	MIN	MAX	AVG
CARDS IN	3	23	10.0
PAGES	1	10	4.8
DATA CARDS	0	10	2.4
LINES OUT	0	16	9.1
TIME	0	248	31.1

Table 5.3 Series II test results

	THRU	TURN	C1	C2	C3	CPU	CORE	DRUM	IB	OB
					% UTILIZATION					
RUN 1	27.1	249	82	74	63	85	75	47	27	58
RUN 2	24.8	326	75	68	57	78	83	56	27	51
RUN 3	24.5	343	74	67	57	76	82	58	26	46
RUN 4	24.8	316	75	67	57	77	83	54	28	45
RUN 5	27.8	252	84	73	62	84	75	47	21	62
RUN 6	27.6	252	83	74	63	85	78	48	35	68
RUN 7	27.5	244	83	73	62	83	72	46	66	73
RUN 8	26.6	268	80	74	63	84	76	51	26	57
RUN 9	26.5	250	80	71	60	81	74	46	26	56
RUN 10	25.6	299	77	69	58	79	81	53	24	49
RUN 11	27.2	262	82	74	63	85	65	49	27	64
RUN 12	18.5	287	93	83	42	57	30	33	11	80

Run 1 is standard and consists of the following configuration:

100 Drum blocks	3 Printer speed	4 MAIN processes
20 Core blocks	3 Reader speed	3 EXECUTE processes
10 Input buffers	.1 Drum speed	1 LOAD process
10 Output buffers		

The following configuration changes are unique to each run:

RUN 1 All channel 3 priorities are equal
RUN 2 Channel 3 priority: Output > Input = Load > Execute
RUN 3 Channel 3 priority: Input > Output = Load > Execute
RUN 4 Channel 3 priority: Load > Output = Input > Execute
RUN 5 Channel 3 priority: Execute > Input = Output > Load
RUN 6 5 input and output buffers
RUN 7 2 input and output buffers
RUN 8 4 EXECUTE processes
RUN 9 2 EXECUTE processes
RUN 10 15 core pages
RUN 11 25 core pages
RUN 12 5 unit reader/printer speed

for allocating time on channel 3 among four types of competing processes. This was done by assigning priorities to the processes so that the channel allocator could choose the process with the highest priority to use the channel next. The only appreciable differences in these runs occur in runs 1 and 5 which have throughput figures which are 9% to 13% higher than runs 2-4. This would indicate that the execution processing node is the critical one with respect to channel 3 utilization and that it should have equal or greater priority than other processes in using this resource.

Runs 6 and 7 reduce the number of input and output buffers, first to 5 and then to 2. As can be seen from the table, the input and output buffer utilization increased as would be expected, but the throughput is virtually unchanged from the standard run. We can thus conclude that two buffers are sufficient for this MOS configuration.

Runs 8 and 9 vary the number of processes in the execution node. In run 8 it is increased to 4 while in run 9 it is decreased to 2. In both cases the throughput is slightly reduced. The run with the reduced number of execution processes exhibits faster turnaround time apparently because the additional time spent in the job queue is more than compensated for by faster execution node processing due to less interference from other user jobs. The reduction in throughput resulting from an increase in the number of execution processes can be explained by the tendency of user jobs to interfere with other system processes through the use of channel 3. Throughput may be reduced by a decrease in the number of execution processes because of the slower flow of jobs through the execution phase. We can thus conclude that three user jobs being processed in parallel is optimal.

The last three runs reflect changes to the hardware of the abstract machine. Runs 10 and 11 test the marginal effect of decreasing or increasing user core storage by 5 core blocks. The decrease in core size reduces system throughput by about 8% while the increase in core size has little influence on throughput. In all cases the maximum core utilization was 100%. These results indicate that 20 core blocks is about optimal for this configuration, assuming 3 execution processes.

The final run changes the reader and printer speeds from 3 time units to 5 time units. As expected the channel utilization is thereby increased for channels 1 and 2 and is reduced for channel 3. The input buffer utilization is decreased (because the buffers are being emptied proportionally faster than they are being filled) and output buffer utilization increases because of the opposite effect. We see that the 2/3 decrease in peripheral speeds reduces throughput by about 1/3.

On balance, the MOS as it is currently configured is remarkably stable and efficient. No significant bottlenecks were discovered and no change to the system increased throughput by more than 3%. Many more experiments could be run to observe performance changes due to system parameters.

Especially of interest would be experiments to test second and higher order effects (changing two or more variables at the same time) to provide a better understanding of overall system behavior.

5.8 MOS program analysis

An analysis of the operating system program for the MOS provides some interesting insight into the proposed methodology. The number of lines of BCPL code is used as a measure of effort expended on the system. This criterion has the advantages of being easily available and should be quite accurate as a qualitative standard, especially when only one programmer is involved in the coding.

Table 5.4 presents a breakdown of the approximately 1200 lines of BCPL code which was required to implement the MOS. Along the left hand side of the table the code is classified according to its level. The level independent classification is reserved for code which does not easily correspond to any level of the simulation.

Along the top of the table we have the four categories: (1) nucleus (NUC), (2) initial program load (IPL), (3) levels, and (4) changes (CHNGS). The nucleus category, in addition to including the primitives for process and resource semaphores mentioned above, includes resource allocators, input/output routines, queue handlers, and many general purpose routines used throughout the operating system. The initial program load includes all the code which is required to read system parameters, spawn the processes of the simulation, and create and initialize resource semaphores. The category labeled "levels" includes all programs for hardware and software processes in the operating system. Finally the "changes"

Table 5.4 Lines of BCPL/360 code

	NUC	IPL	LEVEL	CHNGS	TOTAL
NO LEVEL	266	37	0	NA	303
LEVEL 1	109	44	61	NA	214
LEVEL 2	23	21	20	3	67
LEVEL 3	122	58	235	2	417
LEVEL 4	39	9	162	34	244
TOTAL	559	169	478	39	1245

classification applies only to levels 2-4 and specifies the number of lines of code that had to be changed in the previous levels to extend the simulation to the indicated level.

One desirable goal in light of such a table would be to have all the row totals about equal so that each level would represent about the same amount of work in the total implementation. If any step is significantly larger than the rest it should be broken down into two steps if possible. These figures for the MOS are reasonably well balanced except for the small number of lines in level 2. The levels should probably not be considered dangerously unbalanced unless they differ by at least an order of magnitude.

The aspect of the table that is most striking is the relatively small figures for the number of lines of code that must be changed from one level to the next. The 39 lines of code that had to be changed represents only slightly more than 3% of the total number of lines of code in the system. This statistic supports the claim that the transitions from simulation to simulation and thus the transition from simulation to implementation costs very little, especially relative to what has been gained by operating in a stepwise fashion.

CONCLUSIONS

The MOS has proved to be a useful tool for testing the ideas presented in Chapters 2, 3, and 4. Because these ideas went through many iterations in the course of this research the operating system was rewritten or partially rewritten several times. For this reason it would be impossible to make comparisons of development time with other implementations, even if there were standards for quality of programmers and time and effort spent on a job.

What can be compared, at least qualitatively, is the structure and the clarity of the systems. What is most striking about this implementation as compared with a dozen others written by groups of students is the uncluttered simplicity of the major processes. Whereas in many other systems the details of process interaction, resource accounting, and data passing appear throughout the system, the implementation described here has suppressed these details through the REQUEST and RELEASE primitives.

The MOS has been an effective means of testing the feasibility of new software ideas. In particular, the resource semaphore concept has been shown to be applicable to a wide variety of synchronization and resource allocation problems in the MOS, and the simulation technique has been used with success in transforming a model of the MOS into the final implementation software. Finally the methodology has served to structure the design process and divide the implementation into a number of smaller, more manageable steps. This MOS implementation is believed to be a good paradigm for further investigations or implementations using these ideas.

CHAPTER 6. CONCLUSIONS

6.1 Evaluation

Let us evaluate this research based on the three contributions mentioned in Chapter 1. In the area of synchronization and resource allocation we have described and implemented a system in which the only process interactions are in the form of a call to one of two primitive operations. The advantage of such a scheme is that it forces processes to communicate in a highly disciplined fashion. This uniformity makes systems easier to understand and debug because it is known that between any two invocations of the primitives a process is logically running. When a process is logically blocked, it is known that it was last executing a REQUEST when it became blocked and that it is currently on the waiting process list of the resource semaphore it requested. The synchronization primitives provide the convenience of having a standard but flexible method of resource organization. With variable queueing routines and allocators we can handle a wide variety of resource organizations while suppressing many of the details of manipulating these resources.

We thus conclude that the resource semaphores and their associated primitives are desirable alternatives to the many diverse special purpose data structures and primitives that are currently used. Their success in the MOS together with the examples in Chapter 2 provide ample evidence for suggesting their adoption in more sophisticated operating systems.

The second contribution is in the area of simulating operating systems. Chapter 3 was devoted to explaining how it would be possible to simulate and implement a system at the same time, using the simulation code as the implementation code. This was done by using an activity-based simulation language or its equivalent such that the processes of the simulation become the processes of the implementation. The hardware and software of the system is originally simulated at a low level of detail and is gradually replaced or augmented by more detailed software or hardware.

At each level the simulated performance is evaluated and used as feedback to the next level of the design. With this evolutionary construction procedure the steps can be kept small and within the capabilities of the implementers.

Also, the simulation provides a wide latitude for testing alternative hardware configurations and software strategies on any of a number of levels of detail. Since the sequence of simulations approaches the implementation in the limit they are guaranteed to be faithful to the implementation. Finally, the multi-level system provides a basis for system documentation on a number of levels and the means for testing modifications to the existing operating system or configuration.

The MOS has served as an interesting and encouraging test study for showing the feasibility of these ideas. They should be verified further by incorporating them into the design and implementation of a small or intermediate sized operating system.

The third area of contribution is perhaps the least well defined of the three and concerns design methodology. It is an attempt to combine the ideas on synchronization and simulation into a cohesive and orderly progression of steps leading to an operating system implementation.

The methodology presented was broken into four stages: systems language implementation, nucleus construction, user job path description, and resource contention resolution. Only the last two phases were described in detail.

In the path description phase the computer system is described in terms of how user jobs traverse the system. The system is simulated at this stage by making estimates of the time spent at each of a number of processing nodes. The number of systems processes used to service jobs at each of the nodes is variable.

In the resource phase of the methodology the resources of the system are gradually added so that processes which had previously waited for unspecified delays, now interact through the resource semaphores and wait for specified resources.

This approach seems to offer several advantages over existing methodologies: the simulation provides feedback at each level, the statistics gathered during the simulation phases are retained in the final implementation to the extent desired, and the idea of first specifying the path of a user job is highly intuitive and very similar to existing procedures.

The methodology applied to the MOS has resulted in a much cleaner implementation than many other implementations written in a haphazard fashion. The MOS experience has shown this to be a feasible approach and one deserving of more rigorous testing.

6.2 Future work

The number of possible refinements or extentions to this work is virtually unlimited. We have only begun to understand the problems of software engineering and as a result, more new problems will appear as the field becomes better defined.

One general field of research concerns the definition and implementation of languages for writing operating systems. We should strive to answer the following questions: Should the language contain a rich set of data structures or should the user be able to define his own? Should the language be machine independent or machine dependent? Should operating systems primitives be incorporated into the systems language or written by the operating system designers? To what extent should such a language be transferrable to different machines?

A second area of research concerns the definition of a nucleus for an operating system. REQUEST and RELEASE have been presented as a pair of primitives for handling all process interactions. We must also find appropriate primitives for handling file systems, for implementing paging systems, time sharing systems, storage protection systems, etc. The major thrust of this effort should be to find basic operations that are applicable for solving these problems in many different environments. The possibility of incorporating some universal primitives as hardware operations also deserves additional study.

A third area which could be pursued to a limited extent is further experimentation with or extensions to the MOS. As was mentioned in Section 5.7, the MOS proved to be disappointingly insensitive to configuration changes. Further experiments could include a dynamic drum channel priority or the testing of second order effects such as changing the amount of core and the number of execute processes simultaneously. Extensions to the system might include non-zero supervisor mode timing, demand paging, a file system, or a multiple level auxiliary storage.

Probably the most fruitful area for research is in evaluation of system performance and evaluation of systems programs. System performance has received a good deal of attention in the last few years. The basic problems are to determine what to measure, how to measure it, and how to use the measurements. In addition, analytical models should be developed to help predict changes in system performance resulting from changes in the hardware or software configuration.

The problem of program evaluation, on the other hand, has received little attention. The problems of program stylistics and program construction is the subject of Dijkstra's "Notes on Structured Programming" [Dij 69] and Wirth's "Program Development by Stepwise Refinement" [Wir 71]. Further research would be desirable in the evaluation of software, not in terms of its run time efficiency, but in terms of its writeability, understandability, and changeability. Not until we can distinguish a good systems program from a bad one will we be able to teach how to program properly.

Clearly the most obvious area for extending this research is its application to a small or intermediate sized operating computer system. This would further test the validity of the ideas presented here and would undoubtealy lead to new developments and refinements. The applicability of the ideas to time sharing and multiprocessor systems should also be investigated.

Software engineering may still be in the rudimentary stages of development, but as more emphasis is placed on the global understanding of systems, an order of magnitude improvement in these systems will be possible. As a result, operating systems will be written faster, will be easier to use and modify, and will be more reliable.

APPENDIX I

SPECIFICATIONS OF THE MOS

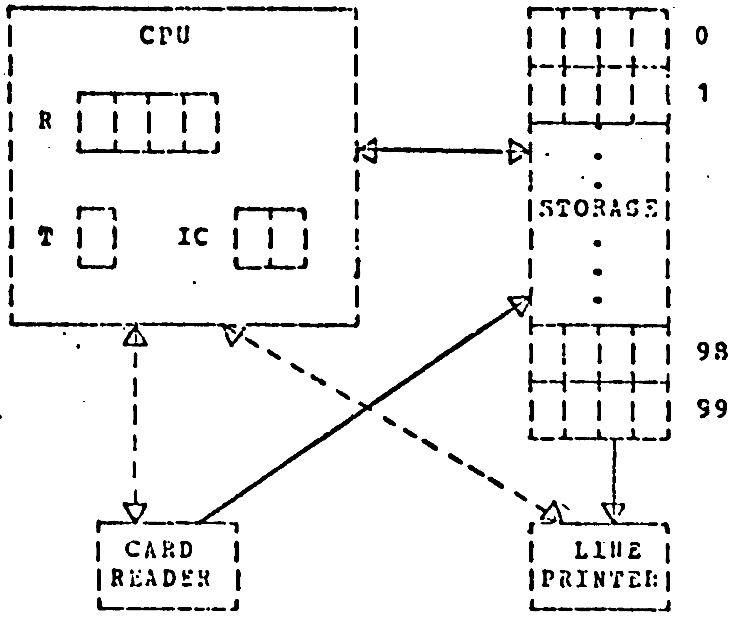
(from [Sha 72])

2. MACHINE SPECIFICATIONS

The virtual machine (VM), that is, the computer as viewed by a normal user, is a very primitive single address computer (Fig. I.1). With the small instruction set, a batch of compute bound, input-output bound, and balanced programs may be quickly written.

The components of the real machine are illustrated in Figure I.2. The CPU may operate in either master or slave mode. In master mode, instructions from supervisor storage are directly processed by the higher level language processor; in slave mode, this processor interprets a "microprogram" in read-only memory which simulates (emulates) the CPU of the VM and accesses VM programs in user storage. This organization allows the VM emulator and the MOS to be coded in a higher level language while maintaining some correspondence with real computers. Currently, the higher level language is BCPL/360 augmented with coroutine primitives [Kel 70; Ric 69; Wil 69]

User storage contains a number of ten-word storage blocks, each word identical to a VM word. User storage is large enough to accommodate several virtual memory spaces to permit multiprogramming of user jobs; a typical size is 30



R: GENERAL REGISTER
IC: INSTRUCTION COUNTER
T: TWO STATE TOGGLE

[ ] : BYTE (CHARACTER)
——— INFORMATION PATH
- - - - CONTROL PATH

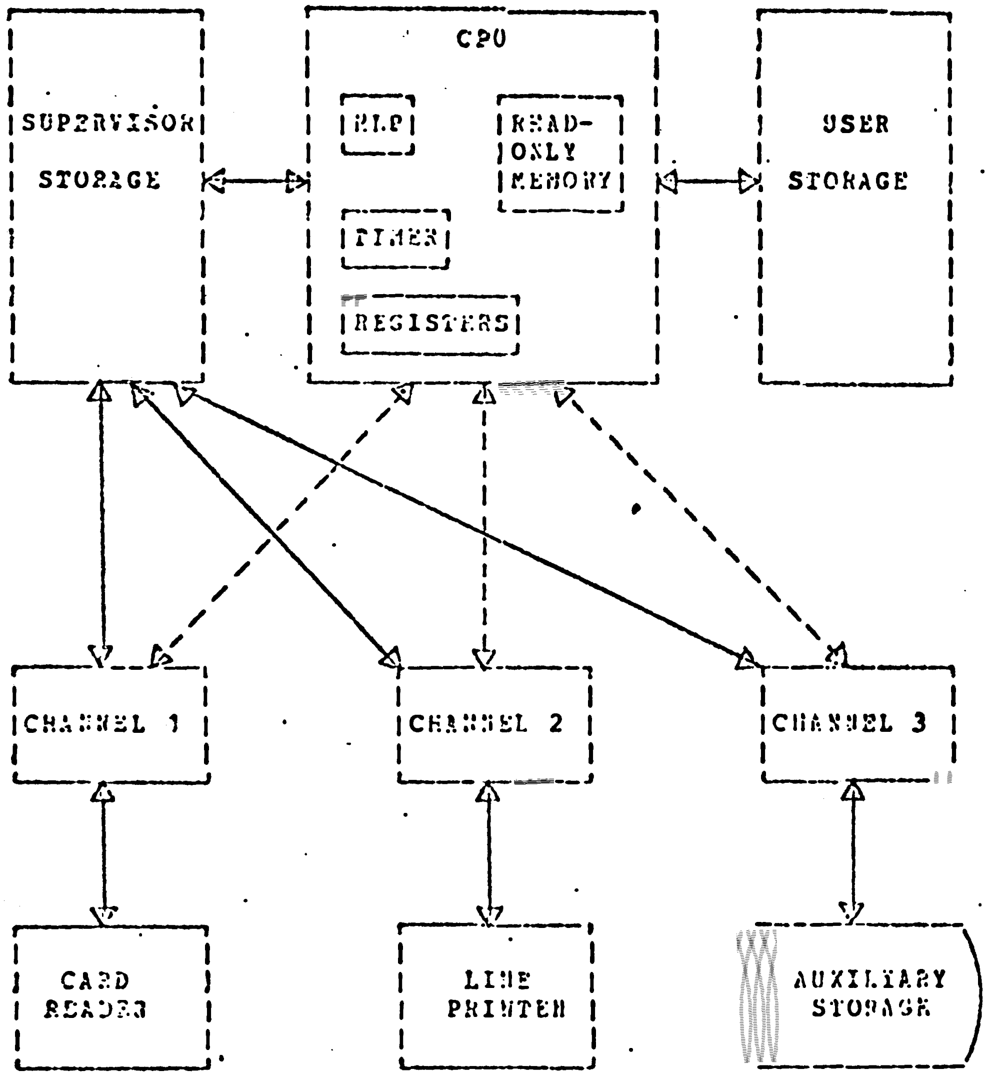
(a) The Virtual Machine

|L|R|1|0| INSTRUCTION WORD

1. Load R (LR)
2. Store R (SR)
3. Compare R to storage (CR)
   If equal then set T; else reset T.
4. Branch if T is set (BT)
5. Read a card (GD)
6. Print a line (PD)
7. Halt (H)

(b) Instruction Set

<u>Figure I.1</u> Specifications of the Virtual Machine



HLP - HIGHER LEVEL LANGUAGE PROCESSOR
----- CONTROL PATH
——— INFORMATION AND CONTROL PATH

Figure I.2 The Real Machine

blocks, but this number may be changed to determine the effects on MOS performance. Addressing occurs through dynamic relocation hardware; a page table register in the CPU points to a page table in user storage associated with the currently executing user process. This permits a VM program to be non-contiguously allocated on a block basis; we do not provide for demand paging. Supervisor storage is loosely defined as that amount of storage required for the MOS.

The auxiliary storage drum is organized in ten-word tracks and is used to store user jobs and output; it typically consists of 100 tracks, but it is again convenient to make this number a parameter. The card reader, printer, and drum are each connected to the system by separate channels which operate independently of each other and the CPU.

Execution times are specified for each of the VM instructions and input-output operations over the channels. In addition, there is an interval timer which may be interrogated and set in master mode. Time runs only while in slave mode with the exception that a master mode program may "wait" for a specified number of time units.

An interrupt mechanism is specified which detects four classes of interrupts: program, supervisor, input-output, and timer. Interrupts occur only at the end of VM instruction cycles in slave mode, but their presence may be tested in master mode. Program interrupts are generated by user storage protection violations and invalid operation codes; supervisor interrupts are caused by a VM read, write, or halt instruction; input-output interrupts result from the completion of a channel operation; and a timer interrupt occurs when the interval timer is decremented to zero. When an interrupt-causing event occurs, its description is stored in a register. The hardware performs the following actions when a slave mode program is interrupted:

(1) save the current slave mode'state,

(2) switch to master mode, and

(3) transfer control to a fixed location in supervisor storage associated with the interrupt class.

3. THE OPERATING SYSTEM

The primary objective of the MOS is to efficiently process a batched stream of user jobs. This is accomplished by multiprogramming system and user processes.

Each user job consists of a job header followed by a VM program and, possibly, data; the job header contains a job identifier, and time and line limits. A job J will pass sequentially through the following phases:

(1) Input spooling

J enters from the card reader and is transferred to the drum.

(2) Main processing

The program part of J is loaded from the drum into user storage. J is then ready to run. Until J terminates either normally or as a result of an error, its status will generally switch many times between:

(a) ready - waiting for the CPU to be allocated to it,

(b) running - executing on the CPU, and

(c) blocked - waiting for completion of an input-output request.

J's input-output requests are translated by the MOS into drum input-output operations.

(3) Output spooling

J's output, including charges, system messages, and its original program, is printed from the drum.

In general, many jobs will simultaneously be in the main processing phase.

The operating system is documented and programmed as a set of cooperating sequential processes [Dij 65; Dij 68]. A typical design would have the following major processes:

ReadInCards:	Read cards into supervisor storage.
JobToDrum:	Create a job descriptor and transfer a job to the drum.
Loader:	Load a job into user storage.
GetPutData:	Process VM input-output instructions.
JobEnd:	Perform a job termination.
PrintLines:	Write output lines on the printer.

A process scheduler, invoked by interrupt handling routines and systems processes, is responsible for allocating the CPU to ready processes.

A major task of the MOS is the management of hardware and software resources; Table I.1 contains the producer-consumer relationships for some of the resources with respect to the above processes. The MOS is also responsible for maintaining statistics on hardware resource utilization and job characteristics. BCPL code for a version of the Loader process is presented in Figure 5.12.

TABLE I.1

PRODUCERS AND CONSUMERS OF TYPICAL RESOURCES[1]

Resource	Consumer	Producer
free buffers[2]	ReadInCards, LinesFromDrum	JobToDrum, PrintLines
card buffers[2]	JobToDrum	ReadInCards
line buffers[2]	PrintLines	LinesFromDrum
free core blocks	Loader	JobEnd
free drum tracks	JobToDrum, GetPutData	GetPutData, LinesFromDrum
drum resident jobs	Loader	JobToDrum
channel 3	JobToDrum, Loader, GetPutData, LinesFromDrum	JobToDrum, Loader, GetPutData, LinesFromDrum

---

1 This tabular form of presentation was originally suggested by T.R. Wilcox.

2 Part of a buffer pool in supervisor storage.

APPENDIX II

SAMPLE OUTPUT FROM THE MOS

(57 STUDENT JOBS AND STANDARD CONFIGURATION)

LEVEL 1 OUTPUT

LEVEL 1:	
JOBS PROCESSED	57
TOTAL RUN TIME	2192
JOBS PER 1000 TIME_UNITS	26.00
PROCESS SWITCHES	308
MEAN TURNAROUND	429.91
MEAN SPOOLIN TIME	30.05
MEAN MAIN WAIT	169.03
MEAN MAIN TIME	35.94
MEAN SPOOLOUT WAIT	67.38
MEAN SPOOLOUT TIME	27.42
MAX MAIN QUEUE	12.00
MAX SPOOLOUT QUEUE	9.00

LEVEL 2 OUTPUT

LEVEL 1:	
JOBS PROCESSED	57
TOTAL RUN TIME	1969
JOBS PER 1000 TIME_UNITS	28.94
PROCESS SWITCHES	642
MEAN TURNAROUND	361.05
MEAN SPOOLIN TIME	30.05
MEAN MAIN WAIT	64.36
MEAN MAIN TIME	58.70
MEAN SPOOLOUT WAIT	123.01
MEAN SPOOLOUT TIME	27.42
MAX MAIN QUEUE	7.00
MAX SPOOLOUT QUEUE	11.00
LEVEL 2:	
MEAN MAIN TIME	
MEAN LOAD WAIT	0.80
MEAN LOAD TIME	4.80
MEAN EXECUTE WAIT	21.95
MEAN EXECUTE TIME	31.14
MAX LOAD QUEUE LENGTH	3.00
MAX EXECUTE QUEUE LENGTH	1.00

LEVEL 3 OUTPUT

LEVEL 1:	
JOBS PROCESSED	57
TOTAL RUN TIME	2206
JOBS PER 1000 TIME UNITS	25.83
PROCESS SWITCHES	11265
MEAN TURNAROUND	434.52
MEAN SPOOLIN TIME	30.07
MEAN MAIN WAIT	202.50
MEAN MAIN TIME	135.47
MEAN SPOOLOUT WAIT	40.84
MEAN SPOOLOUT TIME	25.63
MAX MAIN QUEUE	13.00
MAX SPOOLOUT QUEUE	6.00
LEVEL 2:	
MEAN MAIN TIME	
MEAN LOAD WAIT	17.77
MEAN LOAD TIME	29.17
MEAN EXECUTE WAIT	3.01
MEAN EXECUTE TIME	85.50
MAX LOAD QUEUE LENGTH	3.00
MAX EXECUTE QUEUE LENGTH	1.00
LEVEL 3:	
MEAN JOB CHARACTERISTICS	
CORE PAGES	6.80
CARDS IN	2.36
LINES OUT	2.92
RUN TIME	27.54
I/O OPERATIONS	4.35
OTHER OPERATIONS	27.21
RESOURCE UTILIZATION	
CHANNEL 1	0.7778
CHANNEL 2	0.7547
CHANNEL 3	0.6219
CPU	0.8155
USER CORE	0.7657
MAX CORE UTILIZATION	1.0000
DRUM	0.4010
MAX DRUM UTILIZATION	0.6600
INPUT BUFFERS	0.1097
OUTPUT BUFFERS	0.5852

LEVEL 4 OUTPUT

LEVEL 1:	
JOBS PROCESSED	57
TOTAL RUN TIME	2075
JOBS PER 1000 TIME UNITS	27.46
PROCESS SWITCHES	12191
MEAN TURNAROUND	358.28
MEAN SPOOLIN TIME	30.07
MEAN MAIN WAIT	134.28
MEAN MAIN TIME	128.96
MEAN SPOOLOUT WAIT	41.12
MEAN SPOOLOUT TIME	23.82
MAX MAIN QUEUE	10.00
MAX SPOOLOUT QUEUE	6.00
LEVEL 2:	
MEAN MAIN TIME	
MEAN LOAD WAIT	19.64
MEAN LOAD TIME	27.87
MEAN EXECUTE WAIT	2.59
MEAN EXECUTE TIME	78.85
MAX LOAD QUEUE LENGTH	3.00
MAX EXECUTE QUEUE LENGTH	1.00
LEVEL 3:	
MEAN JOB CHARACTERISTICS	
CORE PAGES	6.80
CARDS IN	2.36
LINES OUT	2.26
RUN TIME	30.87
I/O OPERATIONS	4.31
OTHER OPERATIONS	26.71
RESOURCE UTILIZATION	
CHANNEL 1	0.8269
CHANNEL 2	0.7474
CHANNEL 3	0.6332
CPU	0.8525
USER CORE	0.7884
MAX CORE UTILIZATION	1.0000
DRUM	0.3393
MAX DRUM UTILIZATION	0.5400
INPUT BUFFERS	0.1229
OUTPUT BUFFERS	0.5300
LEVEL 4:	
TOTAL INTERRUPTS	2688
IO INTERRUPTS	2403
SUPERVISOR INTERRUPTS	261
PROGRAM INTERRUPTS	16
TIMER INTERRUPTS	8
PROCESS SWITCHES PER INSTRUCTION	[illegible]

REFERENCES

[ACM 67] Association for Computing Machinery. Proceedings of the Symposium on Operating Systems Principles, October 1-4, 1967, Gatlinburg, Tenn. ACM, 1133 Avenue of the Americas, New York, N.Y.

[ACM 69] Association for Computing Machinery. Proceedings of the Second Symposium on Operating Systems Principles, October 20-22, 1969, Princeton U. ACM, 1133 Avenue of the Americas, New York, N.Y.

[And 69] Androsciani, V., Dickens, C.R., Johnson, T.Y., and Lonergan, R. Evaluation of systems performance in a multi-programming environment. SLAC-PUB-695, Stanford Linear Accelerator Center, Stanford U. Menlo Park, California, Nov. 1969.

[Ber 69] Bernstein, A.J., Detlefsen, G.D., and Kerr, R.H. Process control and communication. In [ACM 69], 60-66.

[Bet 69] Betourne, C. _et.al._ Process management and resource sharing in the multiaccess system "ESOPE." In [ACM 69], 67-74.

[Bri 69] Brinch Hansen, P. RC 4000 software multiprogramming system. A/S Regnecentralen, Copenhagen, April 1969.

[Bri 70] Brinch Hansen, P. The nucleus of a multiprogramming system. _Comm. ACM_ 13,4(April 1970), 238-250.

[Bux 70] Buxton, J.N. and Randell, B. (eds.) _Software Engineering Techniques._ Report on a conference sponsored by the NATO Science Committee, NATO Scientific Affairs Division, Brussels, Belgium, April 1970.

[Com 70] Computer Decisions. How to find bottlenecks in computer traffic. _Computer Decisions_ 2,4(April 1970), 44-48.

[Con 68] Conway, M.E. How do committees invent? _Datamation_ 14,4(April 1968), 28-31.

[Cor 69] Corbato, F.J. PL/I as a tool for systems programming. _Datamation_ 15,5(May 1969), 68,73-76.

[Dah 66] Dahl, O.J. and Nygaard, K. SIMULA - An ALGOL-based simulation language. _Comm. ACM_ 9,9(Sept. 1966), 671-678.

[Dal 65] Daley, R.C. and Neumann, P.G. A general purpose file system for secondary storage. Proc. AFIPS 1965 FJCC, Vol. 27, Part 1, Spartan Books, New York, 1965, 213-230.

[Dij 65] Dijkstra, E.W. Cooperating sequential processes. EWD123, Technological U., Eindhoven, The Netherlands, Sept. 1965.

[Dij 68] Dijkstra, E.W. The structure of "THE" - multiprogramming system. _Comm. ACM_ 11,5(May 1968, 341-346.

[Dij 69] Dijkstra, E.W. Notes on structured programming. EWD 249, Technological U., Eindhoven, The Netherlands, Aug. 1969.

[Efr 64] Efron, R. and Gordon, G. A general purpose digital simulation and examples of its application, Part I: Description of the simulator. _IBM Systems Journal_ 3,1(1964), 22-34.

[Fre 68] Freiberger, I.F. The dynamic behavior of programs. Proc. AFIPS 1968 FJCC, Vol 33, Pt. 2, Thompson Book Co., Washington, D.C., 1968, 1163-1167.

[Hab 69] Habermann, A.N. Prevention of system deadlocks. _Comm. ACM_ 12,7(July 1969), 373-377,385.

[Hol 71] Holt, R.C. On deadlock in computer systems. Ph.D. thesis, Dept. of Computer Science, Cornell U. Ithaca, N.Y., Jan. 1971.

[IBM 68a] IBM System/360 operating system: Supervisor and data management services. IBM form no. C28-6646-2, IBM Corp., Nov. 1968.

[IBM 68b] IBM System/360 Attached Support Processor System (ASP) (360-cx-15x) Version 2: System manual. IBM form no. Y20-0305-0, IBM Corp., Dec. 1968.

[IBM 68c] IBM System/360 principles of operation. IBM form no. A22-6821-5, IBM Corp., 1968.

[Ive 62] Iverson, K.E. _A Programming Language._ John Wiley & Sons, Inc., New York, 1962.

[Kel 70] Kelly, J.R. and Wilcox, T.R. BCPL/360 reference manual. Dept. of Computer Science, Cornell U., Ithaca, N.Y., 1970.

[Knu 64a] Knuth, D.E. and McNeley, J.L. SOL - A symbolic language for general purpose simulation. _Transactions of the IEEE_ EC-13,4(1964), 401-408.

[Knu 64b] Knuth, D.E. and McNeley, J.L. A formal definition of SOL. _Transactions of the IEEE_ EC-13,4(1964), 409-414.

[Knu 68] Knuth, D.E. _The Art of Computer Programming Volume I / Fundamental Algorithms._ Addison-Wesley Publishing Co. Reading, Mass., 1968.

[Kol 71] Kolence, K.W. A software view of measurement tools. _Datamation_ 17,1(Jan 1, 1971), 32-38.

[Lam 68] Lampson, B.W. A scheduling philosophy for multiprocessing systems. _Comm. ACM_ 11,5(May 1968), 347-360.

[Lan 69] Lang, C.A. SAL - Systems Assembly Language. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., 1969, 557-566.

[Lan 70] Lang, C.A. Languages for writing systems programs. In [Bux 70], 101-106.

[Len 64] Leonard, G.F. and Goodroe, J.R. An environment for an operating system. Proc. ACM 19th Natl. Conf., Aug. 1964, E2.3.

[Len 66] Leonard G.F. and Goodroe, J.R. More on extensible machines. _Comm. ACM_ 9,3(March 1966), 183-188.

[Mad 69] Madnick, S.E. and Alsop, J.W. A modular approach to file system design. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., 1969, 1-13.

[Mar 63] Markowitz, H., Hausner, B. and Karr, H. _Simscript: A Simulation Programming Language._ Prentice Hall, Englewood Cliffs, N.J., 1963.

[Nau 69] Naur, P. and Randell, B. (eds.) _Software Engineering._ Report on a conference sponsored by the NATO Scientific Affairs Division, Brussels, Belgium, January 1969.

[Nie 67] Nielsen, N.R. The simulation of time sharing systems. _Comm. ACM_ 10,7(July 1967), 397-412.

[Par 67] Parnas, D.L. and Darringer, J.A. SODAS and a methodology for system design. Proc. AFIPS 1967 FJCC, Vol. 31, Thompson Book Co., Washington D.C. 1967, 449-474.

[Par 69] Parnas, D.L. More on simulation languages and design methodologies for computer systems. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., 1969, 739-743.

[Ran 68] Randell, B. Computer and operating system modeling. VA43, Draft of a lecture given at the University of Utah, Thomas J. Watson Research Center, Jan. 1968.

[Ran 69] Randell, B. Towards a methodology of computing system design. In [Bux 70].

[Rap 68] Rappaport, R.I. Implementing multiprocessing primitives in a multiplexed computer system. Masters thesis (MAC-TR-55), Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., Nov. 1968.

[Ric 69] Richards, M. BCPL: A tool for compiler writing and system programming. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., 1969, 557-566.

[Sal 66] Saltzer, J.H. Traffic control in a multiplexed computer system. Ph.D. thesis (MAC-TR-30), Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., July 1966.

[Sha 68] Shaw, A.C. A multiprogramming system. CS413, Dept. of Computer Science, Cornell U., Ithaca, N.Y., Fall 1968.

[Sha 71] Shaw, A.C. Logical design of operating systems. Notes for CS413, Dept. of Computer Science, Cornell U., Ithaca, N.Y., 1971 (to be published).

[Sha 72] Shaw, A.C. and Weiderman, N.H. A multiprogramming system for education and research. To be published in Proc. IFIP 71 Cong., North-Holland Publishing Co., Amsterdam, 1972.

[Spi 69] Spier, M.J. and Organick, E.I. The Multics interprocess communication facility. In [ACM 69], 83-91.

[Wil 69] Wilcox, T.R. BCPL coroutine primitives. Dept. of Computer Science, Cornell U. Ithaca, N.Y., Fall 1969.

[Wir 68] Wirth, N. PL360, a programming language for the 360 computers. _Journal of the ACM_ 15,1(Jan. 1968), 37-74.

[Wir 69] Wirth, N. On multiprogramming, machine coding, and computer organization. _Comm. ACM_ 12,9(Sept. 1969), 489-498.

[Wir 71] Wirth, N. Program development by stepwise refinement. _Comm. ACM_ 14,4(April 1971), 221-227.

[Wul 70] Wulf, W.A. et.al. BLISS Reference Manual. Computer Science Dept., Carnegie-Mellon U., Pittsburgh, Pa., Jan. 1970.

[Zur 68] Zurcher, F.W. and Randell, B. Iterative multi-level modeling - A methodology for computer system design. Proc. IFIP Congress 68, North-Holland Publishing Co., D138-142.