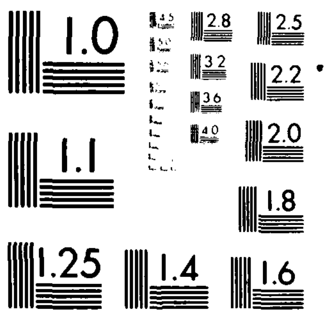
AD-A092 301   NAVAL POSTGRADUATE SCHOOL MONTEREY CA   F/G 9/2
DETAILED DESIGN OF THE KERNEL OF A REAL-TIME MULTIPROCESSOR OPE--ETC(U)
JUN 80 W J WASSON

UNCLASSIFIED   NL

1 of 2
AD
A092301


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

LEVEL II

(2)

B.S.

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

DETAILED DESIGN OF THE KERNEL OF A REAL-TIME MULTIPROCESSOR OPERATING SYSTEM

by

Warren James Wasson

June 1980

Thesis Advisor: U. R. Kodres

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A092 301	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Detailed Design of the Kernel of a Real-Time Multiprocessor Operating System.		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1980
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Warren James/Wasson		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1980
		13. NUMBER OF PAGES 136
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Multiprocessing, parallel processing, operating system, kernel, multiprogramming, processor multiplexing.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis describes the detailed design of a distributed operating system for a real-time, microcomputer based multiprocessor system.
Process structuring and segmented address spaces comprise the central concepts around which this system is built. The system particularly supports applications where processing is partitioned into a set of multiple processes. One such

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
(Page 1) S/N 0102-014-6601

251450

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (Continuation of abstract)

area is that of digital signal processing for which this system has been specifically developed.

The operating system is hierarchically structured to logically distribute its functions in each process. This and loop-free properties of the design allow for the physical distribution of system code and data amongst the microcomputers. In a multiprocessor configuration, this physical distribution minimized system bus contention and lays the foundation for dynamic reconfiguration.

Detailed Design of the Kernel of a Real-Time Multiprocessor Operating System

by

Warren J. Wasson
Lieutenant, United States Navy
B.S., United States Naval Academy, 1975

Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1980

Author: Warren J. Wasson

Approved by:

Uno R. Kodres
Thesis Advisor

Roger R. Schell
Second Reader

[illegible]
Chairman, Department of Computer Science

[illegible]
Dean of Information and Policy Sciences

ABSTRACT

This thesis describes the detailed design of a distributed operating system for a real-time, microcomputer based multiprocessor system.

Process structuring and segmented address spaces comprise the central concepts around which this system is built. The system particularly supports applications where processing is partitioned into a set of multiple processes. One such area is that of digital signal processing for which this system has been specifically developed.

The operating system is hierarchically structured to logically distribute its functions in each process. This and loop-free properties of the design allow for the physical distribution of system code and data amongst the microcomputers. In a multiprocessor configuration, this physical distribution minimizes system bus contention and lays the foundation for dynamic reconfiguration.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. DISCUSSION

The topic of this thesis is the detailed design of the kernel of a real-time microcomputer based multiprocessor operating system. The kernel comprises a complete, albeit primitive, operating system providing support for a large number of asynchronous processes.

The kernel manages all physical processor resources thereby providing the user with an execution environment relatively free from concern about the underlying hardware configuration. The system is capable of performing in a real-time environment through the use of preemptive scheduling to ensure expeditious handling of time-critical processing requirements.

Despite the rapidly expanding capabilities of modern microcomputer systems, they still prove to be limited by the relatively slow execution speeds of their microprocessors. These systems generally do not provide the power and flexibility required to address complex and demanding applications. One such area is that of real-time digital image processing. This is a particularly demanding application area characterized by the requirement to apply significant processing power to a high input data rate.

A natural answer to the inadequacies of the lone microcomputer is to provide for multiple microcomputer systems. Such systems could provide the processing power to adequately handle applications which are presently addressed only within the domain of minicomputers and mainframe systems. However, the general purpose microcomputer operating system which would control such a system does not exist today. Most of today's microcomputer operating systems deal only with uniprocessors and, in fact, could not adequately manage multiple processors.

The integration of large numbers of relatively inexpensive microcomputers into powerful computer systems has been the subject of intensive research in universities and industry for several years. As a result, a number of multiple microcomputer systems such as Carnegie-Mellon's Cm* [18] have been built and even more such as the varied architectures of Anderson and Jensen [1] have been suggested. The Cm* is an ambitious system with 50 processors and a complex, custom designed and built bus structure [18]. Most of the proposed systems require this type of specialized hardware. The primary thrust of this thesis is towards a general control structure which can be applied to hardware systems that are commercially available today with only very minor or no hardware development. Thus no serious attempt is made to consider alternative hardware architectures as a topic in this research.

A complete high level operating system design was provided by O'Connell and Richardson [11] in their family of secure multiprocessor operating systems. This thesis concerns itself with the detailing of one member of their family, a modified real-time subset. The modification consists of the inclusion of a more general synchronization mechanism, eventcounts and sequencers described by Reed and Kanodia [13] which replace the more traditional Signal/Wait and Block/Wakeup used in the original design.

The system supports multiple asynchronous processes using the concept of two-level traffic control to accomplish processor multiplexing amongst a greater number of eligible processes. This dual-level processor multiplexing design allows the system to treat the two primary scheduling decisions, viz., the scheduling of processes and the management of processors at two separate levels of abstraction.

B. STRUCTURE OF THE THESIS

Chapter II describes the overall design philosophy of the operating system, how multiple processes are synchronized and how their multiplexing on a smaller set of processors is accomplished. Chapter 3 describes the hardware architecture of the multiprocessor system in terms of the particular hardware suite chosen for this system. Chapter IV discusses the details of the kernel design. The final chapter presents conclusions and observations that resulted from this effort and suggestions for further research. Two appendices are also provided, an explanation of programming methodology for this system and a detailed description of the kernel modules in their present form.

II. FUNDAMENTAL DESIGN CONCEPTS

A. DESIGN PHILOSOPHY

Multiple processor systems are intrinsically more complex then the familiar uniprocessor. Their complexity has proven to be the major barrier to realizing the full potential of the inherent parallelism available in such a system.

One of the most important components of any computer system is the operating system. The operating system manages the system's resources. Thus system performance is critically dependent upon its effectiveness. However, performance is not just raw computational speed, but is in reality the sum-total of numerous attributes. Some of these system attributes such as ease of programming, correct operation, and the ability to address diverse applications are as important as speed and efficiency, but too often are overlooked. Because of this potentially very large set of requirements, adequate performance can only be assured if the behavior of the system is well understood by the designer. Of necessity, this imposes a strict requirement for simplicity.

In this design, the requirement for simplicity is satisfied by utilizing a model based on the notion of multiple asynchronous processes with segmented address spaces. This is the central unifying concept which provides a straightforward view of both static and dynamic system behavior [4]. The principles of structured system design are also applied to logically organize the operating system into a hierarchically structured set of easily understood modules whose interactions are clearly specified and strictly enforced.

The result is a modular, layered operating sytem which is both smaller and easier to analyze. This, in turn makes it easier to ensure correct operation and provides better opportunity for improving performance through tuning. Certain other benefits accrue from simplificaton as well. Because the sytem is smaller, less memory is used for operating system code and less processor time is spent in its execution.

B. SEQUENTIAL PROCESSES

1. Definition of a Process

The concept of a process has proven to be a fundamental and powerful one in the organization of computer systems. The rather abstract idea of a process has been defined in numerous ways, but perhaps the simplest is offered by J. Saltzer as:

"...basically a program in execution on a processor." [17]

In considering the above definition, it becomes apparent that there are two elements which together completely characterize a given process. They are 1) the program, consisting of any sequentially executed machine instructions and data which can be associated with the program (usually termed the process' address space) and, 2) the execution state of the process which is characterized by the contents of certain processor registers.

2. The Process Address Space

The address space, simplistically, provides for the encapsulation of a process such that it has no knowledge of any other process and no other process has knowledge of it. This eliminates the possibility of inter-process interference simply because processes are unable to "escape" the confines of their defined address spaces.

However, this is rather restrictive in that processes which are totally ignorant of each other have no hope of co-operating towards the accomplishment of some greater goal. In order to mediate this constraint, one desires to allow some restricted (controlled) form of address space overlap (viz., sharing) such that co-operation is allowed while still retaining the benefits of protection offered by isolation. Sharing requires some way of distinguishing the shared portions of the address space. This is greatly facilitated by introducing the notion of memory segmentation.

a. Virtual Memory and Segmentation

Virtual memory is used to implement the concept of a per process address space. In Multics [2], each process is provided with its own virtual memory for an address space. These virtual memories are completely independent of one another.

A virtual memory consists of a set of segments. Segments are distinct variable size memory objects which contain information. Associated with a segment is a set of logical attributes used to uniquely identify the segment and to control access to it.

In specifying the set of segments that comprise a virtual memory, one may include segments that are part of other virtual memories as well. Thus segments can be shared in a controlled manner to provide for inter-process communication and co-operation.

By using segmentation to provide a virtual memory environment, the user is presented with a configuration independent system in that he "sees" a process address space that he can consider his own and is not dependent on the assignment of physical addresses.

b. Addressing in a Segmented System

Addressing in a segmented memory system is two-dimensional. That is, a complete address consists of two parts. The first is the segment number. This identifies the particular segment of interest. One attribute of the segment is the physical address of the segment's base. Thus the segment can be located anywhere in physical memory by changing the base address. The second dimension of the address is an offset relative to the segment's base (the beginning of the segment). This serves to access specific locations within the segment.

C. INTER-PROCESS SYNCHRONIZATION AND COMMUNICATION

Utilizing the parallelism afforded by multiple processors requires a mechanism for inter-process communication and synchronization. It is used for controlling the execution of processes and coordinating the sharing of data.

The most widely used synchronization primitives are Dijkstra's semaphores [3] or Saltzer's Block and Wakeup [17] which were used in O'Connell and Richardson's original design [11]. However, the design decision was made to use a different mechanism which addresses the questions of confinement in a secure system. This is the synchronization mechanism based on the eventcounts and sequencers of Reed and Kanodia [13].

D. PROCESSOR MULTIPLEXING

1. Definition of Processor Multiplexing

Processor multiplexing is a technique for sharing scarce processor resources among an arbitrarily large number of processes. It is accomplished by simulating the existence of a larger number of virtual processors. This technique is widely used in conventional uniprocessor systems where it is commonly called multiprogramming. It seeks to maximize the use of the available hardware by automating control of process loading and execution. It also greatly increases the flexibility of a system allowing it to be effective in more complex and demanding applications.

J. H. Saltzer [17] presented one of the fundamental works on the subject of processor multiplexing. His thesis provides an excellent treatise of the salient issues.

2. Processor Virtualization

In order to effect processor multiplexing, the physical processor resources (those hardware devices that execute machine instructions) are virtualized by creating abstract processors called virtual processors.

a. Virtual Processors

Each physical processor posseses some internal memory (registers) whose contents describe the processor's state. As part of the processor state, there is a specification of the accessible address space which contains the instructions and data used by the processor.

Virtual processors are simulations of processors. They can be viewed in essentially the same way as physical processors in that they execute the same instructions. However, the instruction set of a virtual processor has been expanded to include some instructions which the physical processors do not directly have. These include "instructions" to "load" a process, certain synchronication primitives, system service calls, etc.

Virtual processors exist only as abstract processors represented by a data structure. They are used as the vehicle for the control and manipulation of processor resources.

3. Two-Level Processor Multiplexing

In this design, there are two levels of processor multiplexing. This design arose from the existence of multiple physical processors. Each of the levels address a distinct requirement. One level supports virtual processor management, that is, the provision of inter-process synchronization. The other supports the management of physical resources by the operating system.

This divides the requirements for multiplexing mechanisms into two parts. One of these addresses multiplexing virtual processors among processes and the other multiplexing physical processors among virtual processors.

a. The Traffic Controller

The Traffic Controller represents the upper level of processor multiplexing (termed Level 2) and provides the mechanism for multiplexing virtual processors among processes. Thus it is responsible for inter-process synchronization.

As an example, consider that a process, called A, will wish to synchronize its actions with another process, called B, such that process B will have to complete some task before A can continue execution. Thus A will execute to the point where it cannot proceed further and wishes to signal process B. When process B has finished its task, it must notify process A of its completion so that process A may then proceed.

This inter-process synchronization is handled at the level of the Traffic Controller. When process A discovered that it could not proceed further, it "gave away" its virtual processor to some process that could run. The Traffic Controller suspended the execution of process A and a new process was bound to the virtual processor. In the same way, when B completes, viz., it has no more work to perform, it will also give its virtual processor away.

b. The Inner Traffic Controller

The Inner Traffic Controller comprises the lower level of processor multiplexing (Level 1) and provides the second set of multiplexing functions. It multiplexes the physical processor among one or more virtual processors. While the virtual processors have identical capabilities, the physical processors may differ in their capabilities, viz., they may have different attached I/O devices, different local memory sizes, etc. The Inner Traffic Controller must manage the physical resources in such a way that the user is unaware of these differences. In particular, the system's interrupt system is managed by the Inner Traffic Controller.

If a user process calls upon some system service, such as disk I/O or I/O for a real-time sensor, it must wait for that service to be completed before it can proceed. The performance of a system service is considered to be part of the requesting processes. However, it may actually be supported by another virtual processor. To control this interaction the Inner Traffic Controller provides the required inter-virtual processor synchronization mechanism. In particular, a physical system interrupt is directly transformed into a synchronization signal to a waiting virtual processor. This structure is particularly important for the support of real-time processing.

4. Processor Multiplexing Strategy

a. Process State Transitions

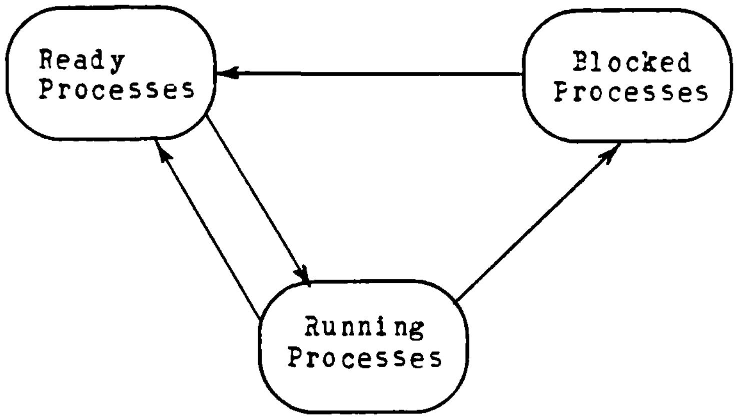
Figure 1 illustrates the state transitions of a set of processes as a virtual processor is multiplexed among them. Some eligible process (one which is in the ready state) is scheduled to run and is bound to the virtual processor. At this time, the process makes the transition to the running state. As far as the process is concerned, once it enters the running state, it is executing.

At some point in its execution, the process may desire to block itself or signal another process. If it blocks itself (enters the blocked state), it will give up the virtual processor to which it is presently bound and will be out of contention for processor resources. It will remain in the blocked state until some other process signals it (thus making the transition back to the ready state). If the process signals other processes, it will transition from the running state back to the ready state from which it may be scheduled to run again. In doing so, it allows the Traffic Controller to possibly give the virtual processor to some higher priority process which may be ready to run.

b. Virtual Processor State Transitions

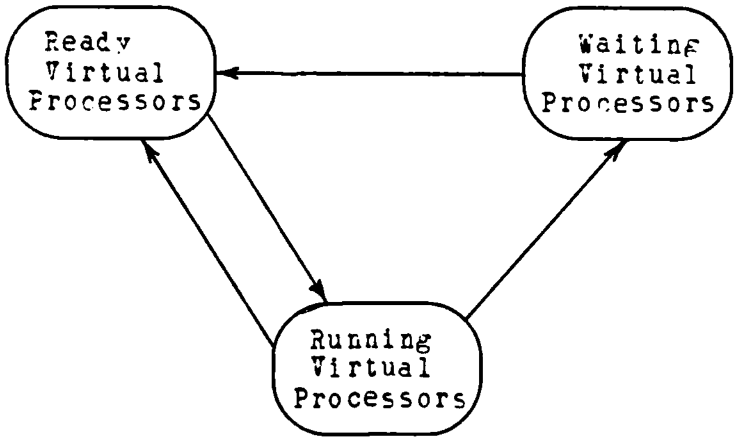
Figure 2 illustrates the state transitions made by virtual processors as a physical processor is multiplexed. This diagram is very similar to that of Figure 1. However, these transitions are not directly observeable by processes (except as differences in execution times) as virtual processor state transitions result from the management of physical resources by the operating system.

In Figure 2, it can be seen that a running virtual processor can transition to the waiting state or the ready state. The transition to the waiting state occurs when a virtual processor must wait for completion of some system service (analogous to the blocking of process A in the example given in paragraph a). While in the waiting state, the virtual processor is out of contention for processor resources until another virtual processor signals it to continue. While in the ready state, the virtual processor is in contention for processor resources and so may be scheduled to run on the physical processor.



PROCESS STATE TRANSITIONS

Figure 1



VIRTUAL PROCESSOR STATE TRANSITIONS

Figure 2

III. MULTIPROCESSOR ARCHITECTURE

A. HARDWARE REQUIREMENTS

One of the principal design goals of the system design was to provide for configuration independence. Therefore, the operating system imposes but a few constraints on the hardware that are noted here.

1. Shared Global Memory

The operating system maintains system-wide control data accessible to each of the processors via shared segments. The communication path utilized for sharing this data is shared memory. Thus some shared memory must be made available to each microcomputer in such a way as to allow independent access at the level of single memory references.

2. Multiprocessor Synchronization Support

There must exist some hardware-supported multiprocessor synchronization primitive. This can be any form of an indivisible read-alter-rewrite memory reference. This capability is required to implement global locks on shared data to prevent race conditions as the physical processors attempt to asynchronously manipulate shared data.

3. Inter-Processor Communication

Some method of communication between physical processors must be provided. This is satisfied by an ability to generate interrupts between the physical processors. This capability is required for the implementation of preemptive scheduling.

B. HARDWARE CONFIGURATION

1. System Configuration

The hardware sub-system is configured as a multiprocessor [1]. The system consists of a number of single board microcomputers and a global memory module connected by a single shared bus. The system differs from conventional multiprocessors in that each of the microcomputers possesses its own local memory. The global memory module is connected directly to the system bus and is the only physical memory resource which is shared by all of the processors. The general configuration is shown schematically in Figure 3.

2. Specific Hardware Employed

The particular hardware selected for this implementation is based on the INTEL 86/12A single board microcomputer [6]. This microcomputer utilizes the INTEL 8086, a 16-bit general-purpose microprocessor capable of directly addressing a total of 1 mega-byte of physical memory.

a. The 8086 Microprocessor

The 8086 does not support the notion of explicit segmentation. In the 8086, addressing is segment-like in that base and offset addressing is used. The offsets are formed relative to one of the four segment base registers of the 8086: 1) the Code Segment Register, used for addressing a pure segment containing executable code, 2) the Data Segment Register, used for process local data, 3) the Stack Segment Register, used for the per process stacks, and 4) and the Extra Segment Register, typically used for external or shared data.

In the 8086, a segment can range anywhere up to 64 kilo-bytes in length. Segments can be placed anywhere within the 1 mega-byte address space of the 8086 as long as the segment base is placed on an even hexadecimal memory address. Segment access and bounds checking are not supported. Although there is no general segmentation hardware, this design effects a segmented address space through a combination of operating system support and system initialization conventions described in a companion thesis by Ross [16].

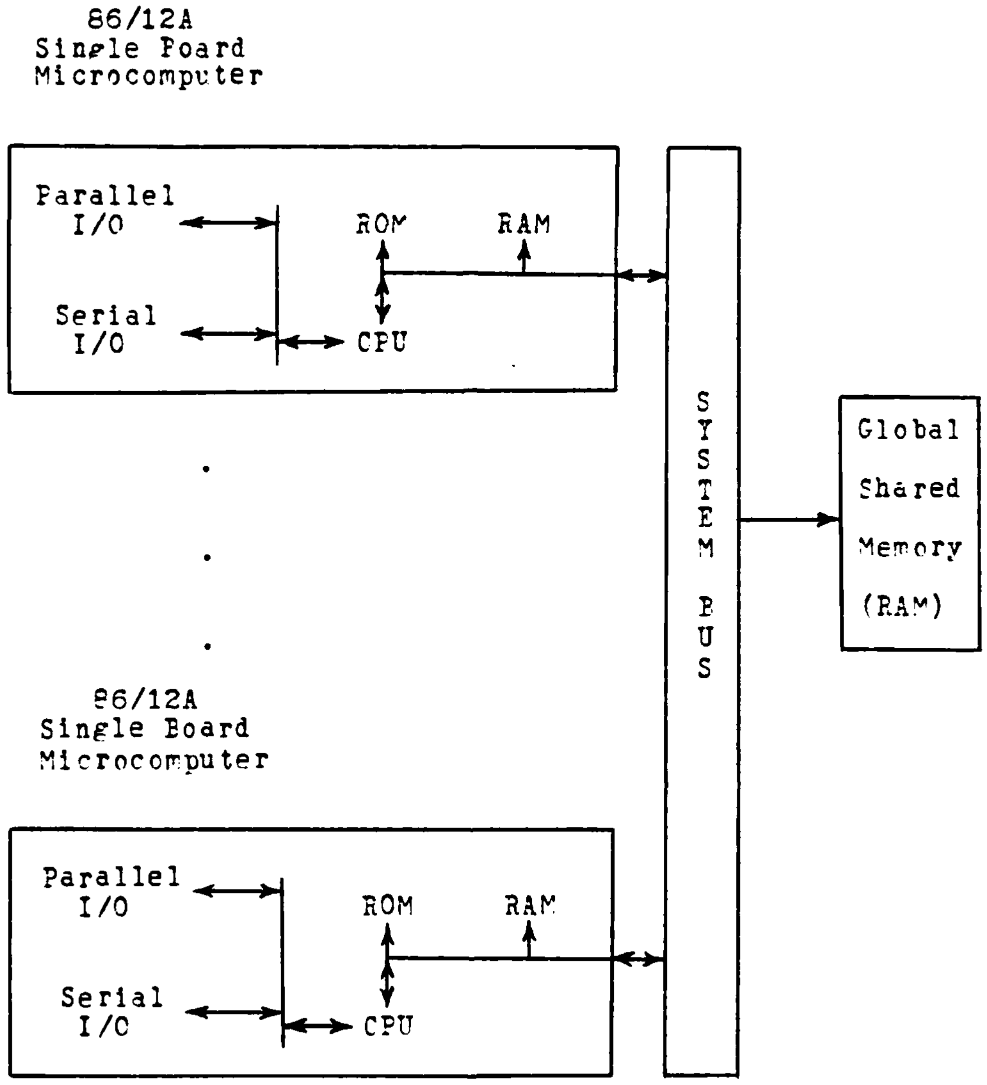
b. The 86/12A Single Board Microcomputer

The 86/12A is a complete computer capable of stand-alone operation used as the basic processing node of the multiprocessor. It is a commercial product which satisfies the three basic hardware requirements for this operating system. First, possessing a system bus interface, each microcomputer is capable of independently accessing a global shared memory via the system bus. Secondly, the 8086 CPU supports multiprocessor synchronizaton directly with an indivisible test-and-set semaphore instruction performed under bus lock. Lock semaphores reside in the shared global memory since the system bus must be locked to ensure that this instruction operates correctly. Thirdly, preempt interrupts can be generated by using the parallel I/O ports provided on each microcomputer. This requires connecting the microcomputer's parallel I/O ports to the system interrupt structure.

c. Preempt Interrupt Hardware Connection

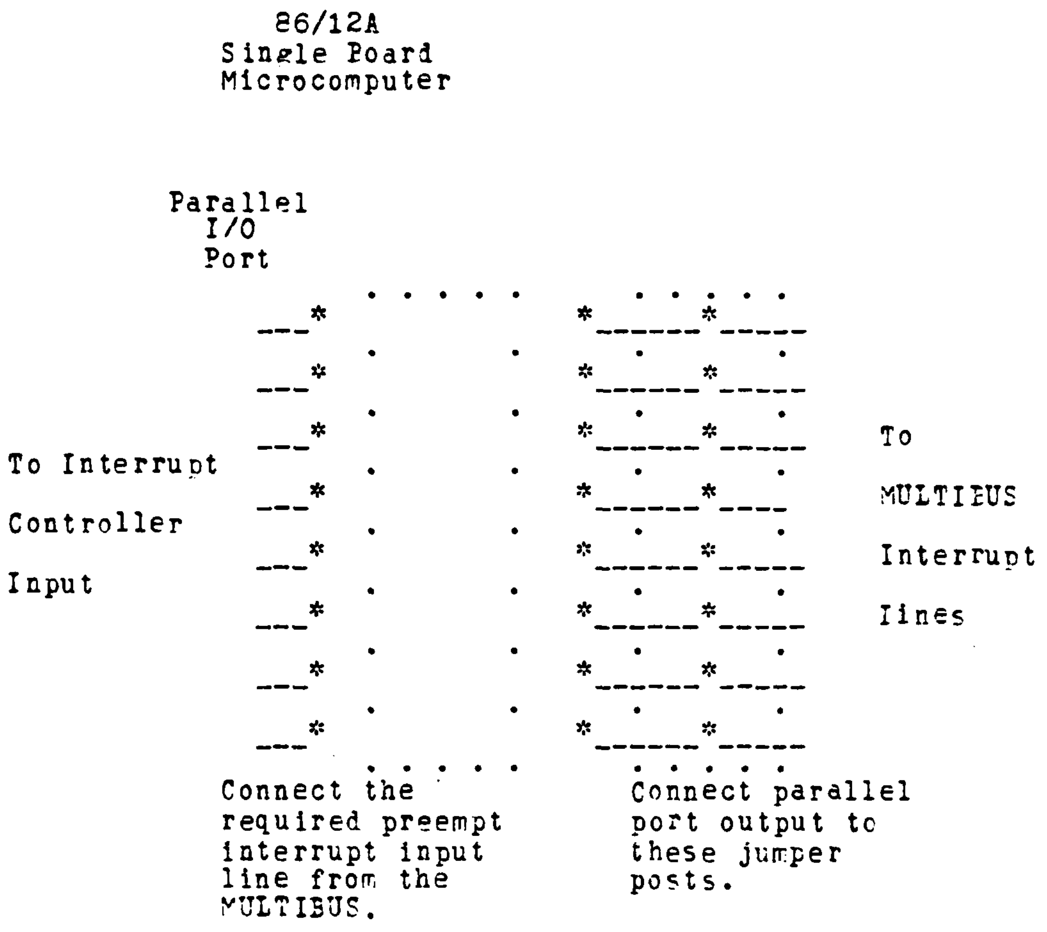
As with most microprocessors, the 8086 itself does not possess the capability to directly generate interrupts destined for other devices (the devices of interest here are other processors). The system interrupt lines are accessible through a jumper matrix [6] located on the microcomputers. The parallel I/O port output of each ISBC 86/12A is connected to this interrupt jumper matrix. Preempt interrupts are then generated simply by outputting a single word through the parallel port onto the system interrupt lines. The connection is shown in Figure 4.

Note that only a single interrupt line is actually required to implement system-wide preempt interrupts. In this implementation, four lines are used. This provides four unique interrupt lines. If more than four processors are used in the system, then these lines are multiplexed (viz., several processors share an interrupt line).



MULTIPROCESSOR CONFIGURATION

Figure 3



PREEMPT INTERRUPT CONNECTION

Figure 4

d. The System Bus

The Intel MULTIBUS [6] is utilized as the system bus. It is a widely used commercial product with a published set of standards. This bus is specifically designed to support multiple processors and is fully compatible with the microcomputers used. It is utilized without modification.

C. HARDWARE ASSESSMENT

The commercially available 86/12A single board microcomputer was chosen because it was specifically designed to provide support for multiple processor systems. In using the operating system described in the next chapter to manage the microcomputer's physical resources, this microcomputer is entirely suitable for use as a basic processing node of an effective multiprocessor system.

IV. DETAILED SYSTEM DESIGN

A. STRUCTURE OF THE OPERATING SYSTEM

This operating system provides a multiprogrammed multiprocessor system with segmented process address spaces using the hardware described in Chapter III. The operating system is structured as a hierarchy of three levels [11], as follows:

Level 3: Supervisor

Level 2: Traffic Controller

Level 1: Inner Traffic Controller

The Inner Traffic Controller (Level 1) forms the bottom level of the hierarchy. It is "closest" to the hardware and encompasses the major machine-dependent aspects of the system. The Inner Traffic Controller multiplexes the physical processor amongst a pool of more numerous virtual processors.

Residing at the next level (Level 2) is the Traffic Controller, which is responsible for multiplexing virtual processors among a larger number of user processes competing for resources. The user-accessible inter-process communication and synchronization primitives (Advance, Await and Ticket) provided at this level allow the user to easily address complex system-wide inter-process synchronization requirements.

The Supervisor resides at the topmost level (level 3). The Supervisor's purpose is to provide common services for user processes. In this implementation, it only provides a simple higher order language interface to the kernel by having a single entry point into the kernel.

B. DISTRIBUTING THE OPERATING SYSTEM

One of the primary concerns in any multiple computer system is the issue of performance. In this type of system, a multiprocessor with a single shared system bus, the most glaring potential bottleneck is the system bus. It then becomes highly desirable to minimize accesses to this resource that must be shared by all of the microcomputers.

In terms of the design, the described system is a distributed operating system patterned after Multics [12]. In particular, the segments of the operating system kernel are distributed as part of the address space of each process. In terms of the implementation of this system, the performance issue is addressed by physically distributing copies of the kernel in the local memories of each of the microcomputers. This allows high-speed access to kernel functions without necessitating use of the system bus for code fetches.

Thus each computing node can be regarded as semi-autonomous in that each of the processors schedule themselves but are still centrally controlled by the set of system-wide data tables. There is no concept of a master-slave relationship among individual microcomputers, nor are individual kernel functions divided up among them as is more often done. Rather the entire kernel is distributed.

C. REAL-TIME PROCESSING

Real-time processing involves the performance of time-critical processing often related to the control of external devices. This application requires that some mechanism be employed to ensure that time-critical processing is given immediate attention.

The hardware-supported process preemption mechanism employed in the system provides the rapid response required for real-time processing. The priority-driven preemptive scheduling technique used provides for expeditious handling of processes which perform time-critical functions. These processes are assigned high priorities so that the system will preempt other processes of lower priority that may be running. Thus when one of these high-priority processes is signalled, it can be immediately scheduled and gain control of processor resources.

D. PROCESS ADDRESS SPACES

The address space of a process is a set of PL/M-86 segments: procedures (code), local variables (data), external data (shared data), and stack [12,13]. Physical memory is allocated to the segments of a process in such a way as to limit system bus contention, as discussed by Ross [16]. In this system, the stack is a key element in the management of processes.

1. <u>The PL/M-86 Stack</u>

Intel's high order language PL/M-86 [5, 16] utilizes stack segments to implement per process stacks. Addressing of stacks is accomplished by using three of the 8086's registers as shown in Figure 5. The Stack Segment (SS) Register contains the base location of the stack segment in memory. The Stack Pointer (SP) Register addresses the current top of the stack as an offset from the base of the stack segment, (the value in the SS Register). The Base Pointer (BP) Register also holds an offset from the SS Register and is used to establish procedure activation records [7, 8, 9].

2. <u>The Stack as the Address Space Descriptor</u>

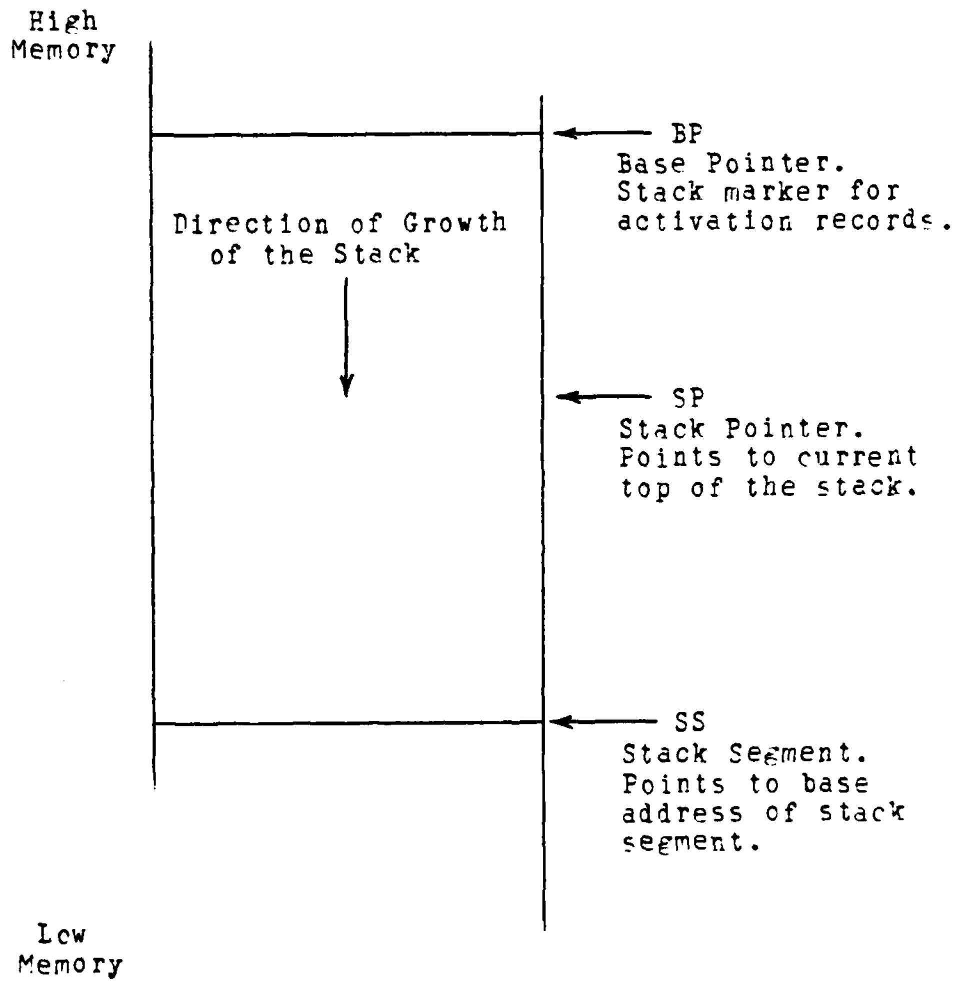
In this system, the per process stacks are used to maintain process state information. This includes the current execution point (when the process is not actually running), the type of return from the kernel required for the process (normal or interrupt) and the locations of the code and data segments. This allows the system to swap in a new address space (viz., do a context switch) by changing the value in the SS Register, which is thus used in a manner somewhat analogous to the Multics Descriptor Base Register [12].

Figure 6 shows how this information is stored in the stack while a process is not actually running on a physical processor. The Base Pointer, Stack Pointer and Return Type Indicator are stored in reserved locations at the very beginning of the stack segment.

In order to identify the stack segment, and thus access the address space of a process, the stack segment base address is used in a dual role. First, a unique base address is assigned to the stack of each process which provides a unique segment for each stack. This base address is used for addressing locations within the stack. Secondly, the base address serves as a descriptor for the address space of each process. Thus the binding of a processor is changed from one process to another "merely" by changing the base address, viz., changing the value in the Stack Segment (SS) Register.

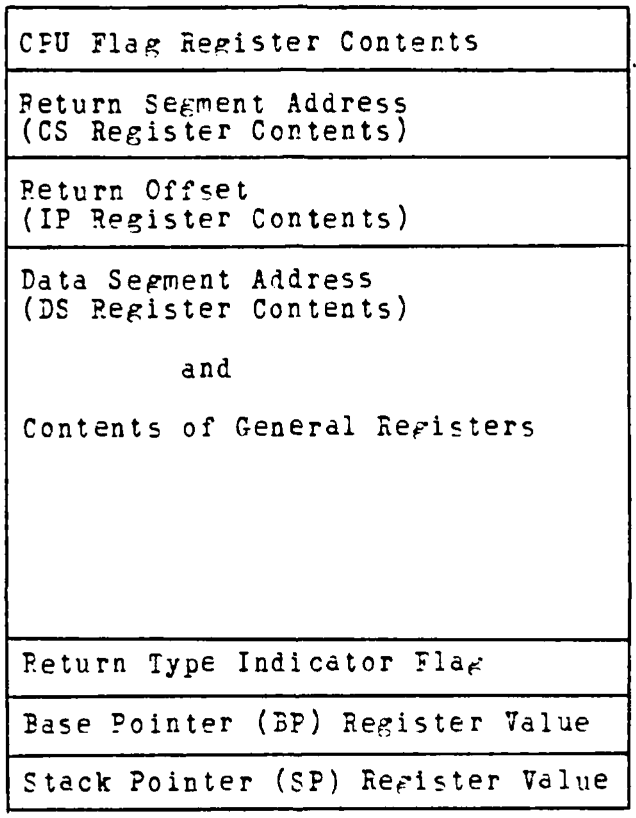
E. SYSTEM PROCESSES

System processes make up the non-distributed kernel. Non-distributed refers to the fact that these processes are not distributed as part of each process' address space. Rather they represent various system services. System processes are used for the management of hardware resources


High Memory
BP
Base Pointer. Stack marker for activation records.
Direction of Growth of the Stack
SP
Stack Pointer. Points to current top of the stack.
SS
Stack Segment. Points to base address of stack segment.
Low Memory

PL/M Stack Structure

Figure 5


CPU Flag Register Contents
Return Segment Address (CS Register Contents)
Return Offset (IP Register Contents)
Data Segment Address (DS Register Contents)
and
Contents of General Registers
Return Type Indicator Flag
Base Pointer (BP) Register Value
Stack Pointer (SP) Register Value

Stack Usage For Process Address Space

Figure 6

and execute asynchronously with respect to user processes. In this design, all system processes are permanently bound to dedicated virtual processors.

1. The Idle Virtual Processor

The idle virtual processor provides the physical processor with a consistent state when no other virtual processor is ready to be run. The idle virtual processor assures that physical processors always have some valid process address space to execute in, although in this case it is only an idle process that performs no useful work.

This is assumed by creating for each physical processor a dedicated idle virtual processor. The idle virtual processors act as "default" that will only be run when no other runnable virtual processors are found.

F. SYNCHRONIZATION

Synchronization is required at two levels in this system: between processes (at the Traffic Controller level) and between virtual processors (at the Inner Traffic Controller level). Both levels use the eventcount and sequencer mechanisms [13] described below.

1. Eventcounts

Eventcounts are used in this system to allow processes to arbitrate access to shared resources.

An eventcount is defined by Reed to be:

"an object in the system that represents a class of events that will eventually occur." [14]

Each eventcount represents a distinct class of events. An eventcount is associated with some type of event of interest, e.g., occurrence of a real-time interrupt, a buffer becoming full, a data segment being read or written into, etc. Eventcounts are implemented as sets of positive integers from 0 to infinity (the limit is actually 65,536 using PL/M-86 "word" variables which is "adequate" for the applications anticipated) and are used to keep track of the total number of such events that have occurred.

Three operations are defined on eventcounts. The value of an eventcount may be obtained by the READ operation. This returns the present value of the eventcount as a positive integer k. From this value, one may infer that events 0 to k have already occurred.

The AWAIT operation allows a process to suspend its own execution (enter the blocked state) until a specified event has occurred, viz., the eventcount reaches the value specified. The effect is the same as the conventional Block operation or Dijkstra's "P" operator.

An ADVANCE operation is performed by a process when an event has occurred. It increments the value of the eventcount by one to reflect the occurrence of the event. This has the effect of signalling the event's occurrence to other processes which were waiting for it by virtue of having previously performed an AWAIT operation. The effect of an ADVANCE operation is essentially the same as a Wakeup operation or Dijkstra's "V" operator.

The eventcount signalling mechanism has an automatic broadcast effect which offers an advantage in parallel processing. This broadcast capability allows the simultaneous signalling of several processes which otherwise would would have to be signalled sequentially.

2. Sequencers

There are many situations where accesses to shared resources must be totally ordered. Eventcounts alone are not sufficient to accomplish this. To provide the capability for mutual exclusion, another type of object called a sequencer [13] is employed. A sequencer is implemented as a positive integer ranging in value from 0 to infinity (as with eventcounts, the limit is 65,536). However, a sequencer is used to provide total order to the occurrence of events. Initially a sequencer has a value of 0. The value increases by one each time a TICKET operation is performed on it. TICKET is the only operation defined on a sequencer. TICKET returns a unique monotonically increasing value with each call. Thus, a set of events can be totally ordered by the TICKET operation.

3. Inter-Process Synchronization

Access to shared resources is easily controlled by using eventcounts and sequencers in concert, as shown in the following "producer/consumer" example [13].

Consider that some hypothetical consumer process called Printer uses a single input buffer in which it finds information to be processed (output to the printer). There are also an unknown number of producer processes called PROD1, PROD2, etc., which have information that they want Printer to output for them. Obviously, with a single buffer, only one of the processes can use the buffer at any one time. The solution uses one sequencer and two eventcounts to properly mediate access to the buffer using mutual exclusion.

The sequencer Turn is used by the producer processes to synchronize their use of the input buffer. The eventcounts Full and Empty are used to synchronize with Printer. Each of the producer processes will execute the program shown below.

```
PROD1, PROD2, etc.   /* Producer programs        */

 DO;
   T = TICKET(TURN); /* Get a "ticket" (turn)    */
                     /* for the buffer           */
   AWAIT(EMPTY,T);   /* Wait for buffer ready    */
          .
          .          /* Write into the buffer    */
          .
   ADVANCE(FULL);    /* Signal Printer that      */
                     /* there is work to do      */
 END; /* DO */
```

Each of the producer processes first performs a TICKET operation on the sequencer Turn to obtain a "ticket" for the buffer. Each time TICKET is called, the variable T of the calling producer process will receive a unique value. This value is then used by the producer process as an argument for the call to AWAIT. It is the event (value of the eventcount EMPTY) for which the process will wait. When that event does occur (the value of Empty, which is advanced by Printer, reaches the value specified in the call to AWAIT) the process will be unblocked and may then proceed to use the buffer. When it has finished, the process will perform an ADVANCE operation on the eventcount Full to signal Printer that there is information in the input buffer. Since each producer process uses the same sequencer, only one of them at a time will access the buffer.

The consumer process Printer is programmed as follows.

```
PRINTER                 /* Consumer program            */

 DO I = 1 TO 65536;     /* Essentially forever         */
   AWAIT(FULL,I);       /* Wait for a message to be    */
                        /* deposited in the buffer     */
              .
              .         /* Perform output function     */
              .
    ADVANCE(EMPTY);     /* Notify waiting processes    */
                        /* that the buffer is now      */
                        /* available                   */
  END; /* DO */
```

The Printer process synchronizes on the eventcount Full (it waits until Full is advanced by some producer process that has finished using the buffer). After Printer finishes with the buffer, it performs an ADVANCE operation on the eventcount Empty. This notifies the producer process that is "next in line" that the buffer is now available for its use.

G. THE INNER TRAFFIC CONTROLLER

1. General Description

The Inner Traffic Controller is the physical resource manager. It is responsible for physical processor multiplexing. Its principal data base is a table known as the Virtual Processor Map.

Each physical processor has its own fixed set of virtual processors used in multiplexing. The Inner Traffic Controller is primarily concerned only with this set of virtual processors. However, the performance of system-wide synchronization requires access to the rest of the virtual processors as well, so that signals may be sent to other physical processors. This is accomplished by maintaining the Virtual Processor Map as a central data base containing entries for all of the virtual processors in the system. Making it globally available facilitates communication between virtual processors on a system-wide scale. The Virtual Processor Map fields are diagrammed in Figure 7.

The State field reflects the present state of the virtual processor and can be any of ready, running, waiting, or idle. A ready virtual processor is bound to a process and

State	Priority	System Eventcount Identifier	System Event Awaited	Stack Segment Register Value	Preempt Pending Flag

THE VIRTUAL PROCESSOR MAP

Figure 7

is in contention for the physical processor. The running virtual processor is that virtual processor which is actually executing a process on the physical processor. The waiting state reflects physical resource management. The idle state is assumed by a virtual processor which has no process bound to it. The idle state prevents the assignment of useless (idle) work to a physical processor.

The Priority field of the virtual processor is used in scheduling. The highest priority runnable virtual processor is selected to run. This priority is determined by the priority of the process bound to the virtual processor.

The System Eventcount Identifier and System Event Awaited fields are used in system level synchronization.

The Stack Segment Register Value field defines the address space of the bound process. It holds the process address space descriptor. The execution state of the process is stored in the stack when the process is not actually running. This is the value which is required to access the address space of the process, viz., it is changed to swap processes.

The Preempt Pending Flag is used for preemptive scheduling. It serves to virtualize a hardware interrupt sent to the physical processor.

2. <u>Virtual Processor Scheduler (Vp_Scheduler)</u>

This module is responsible for making the scheduling decisions for virtual processors. It selects the highest priority virtual processor from among the physical processor's assigned set of virtual processors and schedules it. Note that there are two distinct entry points to Vp_Scheduler.

The normal call entry point is used by other Inner Traffic Controller modules to activate Vp_Scheduler when a virtual processor gives up the physical processor on its own. The preempt interrupt entry point is used in response to a hardware preempt interrupt from another physical processor.

For a normal call, Vp_Scheduler sets the Vp_Scheduler return type flag to indicate that a normal call-return sequence is to be followed for the executing process. The Vp_Scheduler return type flag is used to keep track of the mode of entry into Vp_Scheduler for the process.

Vp_scheduler next searches through the fixed set of virtual processors for the highest priority ready virtual processor. In this design, the definition of ready includes the combination of an idle state and a pending virtual preempt interrupt. This allows an idle virtual processor to run so that it may field the interrupt and bind to a new process. The idle process that was bound to the virtual processor was essentially useless up until this point. It now provides an address space for the virtual processor to execute in when binding to a new process.

Having selected some eligible virtual processor, Vp_Scheduler proceeds to bind the selected virtual processor to the physical processor. It begins by unbinding the currently running virtual processor. In doing so, the Vp_Scheduler return type flag, the Stack Pointer Register value, and the Base Pointer Register value are saved in known locations on the process' stack. The process' execution state had already been saved.

Binding the selected virtual processor is begun by changing the Stack Segment (SS) Register value to that of the selected virtual processor. Once this change has been made, execution has actually swapped to the new process address space. Binding is completed by retrieving the previously saved Vp_Scheduler return type flag for the new process, the Stack Pointer Register value, and the Base Pointer Register value from the newly acquired stack.

The last step is to actually check the Vp_Scheduler return type flag to determine the proper type of return to execute from Vp_Scheduler for this process. If a normal call-return is indicated, a normal return will be executed back through the calling module. Otherwise, if a preempt interrupt return is indicated, an interrupt return will be executed and Check_Preempt will see if a virtual preempt interrupt is pending. If a preempt interrupt is found to be pending, the Traffic Controller's preempt handler will be invoked.

a. Internal Modules

There is one internal module for the Virtual Processor Scheduler (Vp_Scheduler). It is used for the generation of hardware preempt interrupts.

(1) Hdwr_Int

This module is called by the Inner Traffic Controller's interface modules Itc_Advance and Send_Preempt. It is called with one argument, a physical processor identifier. It then generates the required hardware interrupt.

3. Inner Traffic Controller Interface Modules

a. Load_Vp

This module performs the binding of a new process to a virtual processor. It is called by the Traffic Controller Scheduler when a process has been selected for the virtual processor. Load_Vp requires two parameters, the priority of the new process and the address space descriptor (the Stack Segment Register value). It then swaps in the new process onto the virtual processor which is currently running. Load_Vp only operates on the virtual processor which is running on the physical processor.

Binding is accomplished by updating the Virtual Processor Map. The Inner Traffic Controller utility function Itc_Ret_Vp is used to obtain the identity of the running virtual processor. When complete, the virtual processor will have a new priority and process address space descriptor.

Load_Vp completes by calling Vp_Scheduler to schedule the virtual processor.

b. Idle_Vp

This function is Load_Vp's counterpart. It is called by the Traffic Controller Scheduler in the event that a runnable process is not found for the virtual processor. In this case the virtual processor will be idled (enter the idle state) and the Idle Process will be bound to it. In the Virtual Processor Map, the virtual processor's state will be marked as idle, the address space descriptor for the Idle Process will be entered in the Address Space of Bound Process field, and the virtual processor will be given a high priority. The idle state ensures that the idle process is not actually run (the virtual processor now has a high priority) by taking the virtual processor entirely out of contention for the physical processor.

At some later point, the virtual processor may be placed back in contention for resources. This will occur when the virtual processor is preempted. With the combination of an idle state and a pending preempt, the virtual processor is treated as a high priority ready virtual processor. This allows the virtual processor to keep busy by expediting its binding to a process.

Lastly Idle_Vp calls Vp_Scheduler in order to give up the physical processor.

c. Itc_Ret_Vp

This is a "utility" function which is used by Inner Traffic Controller and Traffic Controller modules. Itc_Ret_Vp searches the Virtual Processor Map and determines the identity of the virtual processor that is currently running on the physical processor. It simply checks for the virtual processor among the virtual processors assigned to the physical processor which is in the running state. Itc_Ret_Vp then returns its result as a function value, (viz., as in PL/M) in the AX (accumulator) register. It will return either the identity of the virtual processor (the virtual processor's index in the Virtual Processor Map) or a "not found" error code.

d. Check_Preempt

This module is called by Vp_Scheduler during the execution of an interrupt return. It checks for a pending preempt interrupt meant for the virtual processor, which has been selected to run by Vp_Scheduler, by checking the virtual processor's Preempt Pending Flag in the Virtual Processor Map. If the Preempt Pending Flag is set, Check_Preempt will reset it and call the Traffic Controller module Tc_Pe_Handler.

The module continuously loops as long as it finds the Preempt Pending Flag set. This is to ensure that a new preempt interrupt which might arrive before servicing of the last preempt is not lost.

e. Send_Preempt

This module is responsible for actually sending preempt interrupts. It is called by the Traffic Controller Advance module. Send_Preempt requires two arguments, the identity of the virtual processor which is to be preempted and the physical processor to which that virtual processor is assigned.

Send_Preempt sets the virtual processor's Preempt Pending Flag and calls Hdwr_Int to generate a hardware interrupt for the physical processor. Hdwr_Int is not called if the virtual processor to be preempted is assigned to the physical processor which is executing Send_Preempt, (viz., a physical processor will not issue a hardware preempt interrupt to itself).

f. Itc_Await

Itc_Await is one of two functions which implements inter-virtual processor synchronization within the kernel. It is not accessible to user processes, but is used by the system in the management of physical resources. It allows a virtual processor to wait for the occurrence of a system event.

It expects two input arguments, the index of the eventcount in the System Eventcount Table and the value of the event to be awaited.

Upon being invoked, Itc_Await locks the Virtual Processor Map. It then checks the current value of the eventcount, obtained from the System Eventcount Table, against the value given in the call. If the present value of the eventcount is found to be less than the value of the input argument, then the virtual processor will enter the waiting state and give up the physical processor.

The virtual processor's entry into the waiting state will be reflected in the Virtual Processor Map. The input arguments will be entered in the Identity of Eventcount Awaited and the Value of Eventcount Awaited fields. Finally, the virtual processor will relinquish the physical processor by calling Vp_Scheduler. Upon a return from Vp_Scheduler, the Virtual Processor Map will be unlocked.

g. Itc_Advance

Itc_Advance is used within the kernel to signal the occurrence of system events. It is used with Itc_Await for synchronization between virtual processors. It accepts one input argument. This is the index in the System Eventcount Table of the eventcount to be advanced.

Upon being invoked, the Virtual Processor Map is locked. The System Eventcount Table is then accessed and the indicated eventcount's value is incremented by one. The resultant value is then compared against the events waited for by other virtual processors which are synchronizing on the same eventcount. Those virtual processors whose Value of

Event Awaited fields are less than or equal to the current value of the eventcount are made ready.

Itc_Advance then calls Vp_Scheduler to schedule the virtual processor. The Virtual Processor Map will be unlocked upon a return from Vp_Scheduler.

H. THE TRAFFIC CONTROLLER

1. General Description

The Traffic Controller manages the execution of user processes. It presents to the user a system of one more virtual processors on which to execute his processes.

The Traffic Controller's primary data base is the Active Process Table, shown in Figure 8. The entry for each process in the Active Process Table contains sufficient information about the process to enable a virtual processor to be bound to and execute it. The fields of the Active Process Table are explained below.

The State of a process can be either ready, running or blocked. A ready process is one which is not yet bound to a virtual processor but is ready to do so. A running process is one which is bound to a virtual processor and, as far as the process is concerned, executing. The blocked state reflects inter-process synchronization. A process enters the blocked state when it realizes that it can no longer proceed and wishes to give up its virtual processor to wait until another process awakens it.

State	Identity of Bound Virtual Processor	Priority	Loaded List Thread	Value of Eventcount Awaited	Block List Thread	Address Space Descriptor

THE ACTIVE PROCESS TABLE

Figure 8

The Affinity field specifies the physical processor that the process must execute on. In this system, this field indicates the specific microcomputer on which the process is currently loaded.

The Identity Of Bound Virtual Processor serves to identify the virtual processor, if any, that the process is currently bound to.

The Priority specifies the priority of the process. In this system, priorities range in value from 0 to 255, with a priority of 0 being the highest.

The Loaded List Thread field serves to implement the Loaded List of ready and running processes. It contains a pointer to the next process in the Active Process Table which is loaded on the same microcomputer as this process. The loaded list is ordered by the priorities of the processes. Thus this field contains either a pointer to a process whose priority is less than or equal to that of this process or a nil pointer (viz., the last process on the Loaded List).

The Value Of Eventcount Awaited reflects the event for which the process has blocked itself. It contains the value that the process is waiting for the eventcount to reach.

The Block List Thread is used to implement the Blocked List. This is a per eventcount list of processes which are waiting on the eventcount.

The Address Space Descriptor field contains the process' address space descriptor. This is the identity of the process' stack which contains execution point information. The value used here is the base location in memory of the stack segment, viz., the Stack Segment (SS) Register value.

2. Process Scheduler (Scheduler)

Scheduler works in essentially the same way that the Inner Traffic Controller's Vp_Scheduler does. However, Scheduler works with processes. Scheduler can be called by Advance, Await, Tc_Pe_Handler, Create_Evc, Create_Seq, and Create_Process.

It selects the highest priority ready process from the microcomputer's Loaded List to be bound to an available virtual processor. Scheduler works only with the processes which are runnable on its own physical processor using the fixed set of virtual processors for that physical processor.

If Scheduler finds a runnable process, the Inner Traffic Controller module Load_Vp is called to bind the selected process to the running virtual processor. Alternatively, if a runnable process is not found, the virtual processor will be idled (bound to the Idle Process and placed in the idle state) by a call to the Inner Traffic Controller module Idle_Vp.

In its present form, Scheduler has only one entry point, a call entry point. There is no interrupt entry point as there is in Vp_Scheduler. This was done as an expedient in this design effort. It is desireable to provide the second entry point so that the two schedulers have parallel structures. Because there is no interrupt entry point, there is a loop between the Inner Traffic Controller and the Traffic Controller for the handling of preempt interrupts. This is due to the call from the Inner Traffic Controller's preempt handler Check_Pre_Empt to the Traffic Controller's preempt handler Tc_Pe_Handler.

a. Internal Modules

There are two "utility" modules internal to the Scheduler that are used only by Traffic Controller modules. They are used to simplify the handling of eventcounts and sequencers.

(1) Locate_Evc. This "utility" returns the index of an eventcount in the Eventcount Table. It is called by Advance, Await and Ticket with (a pointer to) the name of the eventcount. Locate_Evc then attempts to match the name given to it with one in the Eventcount Table. If a match is found, it returns the index to the caller in the AX (Accumulator) Register as a function value (viz., as in PL/M).

(2) Locate_Seq. This is the second Traffic Controller "utility" function. It works in exactly the same way that Locate_Evc does except that it searches for sequencers in the Sequencer Table rather than eventcounts.

3. Traffic Controller Interface Modules

a. Await

Await allows a process to suspend its execution pending the occurrence of a specified event. AWAIT is called with two arguments, (a pointer to) the name of the eventcount and the value (of the event) to be awaited.

Upon invokation, Await locks the Active Process Table and then calls the Inner Traffic Controller utility function Itc_Ret_Vp to obtain the identity of the running virtual processor. This is used in a search of the Active Process Table to identify the calling process.

Once the calling process has been identified, the current value of the eventcount is compared to the awaited value specified in the call. If the event has not yet occurred, (viz., the current value is less than the value to be awaited), then the process will enter the blocked state. The Value of Eventcount Awaited field in the Active Process Table is updated with the value awaited argument and the process is placed on the eventcount's Blocked List. If the event has already occurred, (viz., the current value is greater than or equal to the value awaited argument), then the process is not blocked but is made ready.

Await next calls Scheduler. The Active Process Table is unlocked upon the return from Scheduler.

b. Advance

Advance allows a process to signal the occurrence of an event. It updates the eventcount and signals those processes which had blocked themselves for this event. Thus Advance is responsible for preemption.

Advance is called with one argument, (a pointer to) the name of the eventcount being advanced.

It first locks the Active Process Table. Then the current value of the eventcount is incremented. The eventcount's Blocked List is searched for processes which had previously blocked themselves for this value. As processes are found that should be awakened, they are made ready. An entry in a temporary array of physical processors is now made to flag the physical processor, in whose local memory the newly awakened process is loaded, for preemption. The awakened process is then removed from the eventcount's Blocked List.

Once all of the processes to be awakened have been found, Advance determines which virtual processors must be preempted. This is done for each of the previously flagged physical processors by first assuming that all of the physical processor's virtual processors should be preempted. Then the decision is made as to which ones will not be preempted. This method greatly simplifies the algorithm. First a temporary list (array) of virtual processors is initialized to indicate a virtual preempt for each of the virtual processors. The Loaded List is then searched to find those processes which should be running. The processes which should be running are those with the highest priorities that are in either the ready or the running states. Assuming there are 2 virtual processors per physical processor used for multiplexing, the 2 highest priority ready or running processes in the Loaded List should be running. Any lower priority processes that actually are running should be preempted. Advance determines which of the processes that should be running already are running and deletes their virtual processors from the preemption list (resets the preempt flag in the array). What will remain at the end are those virtual processors that are to be preempted.

The next step is to actually issue the preempt interrupts. The temporary preempt list is checked and if a preempt is indicated for a virtual processor, the Inner Traffic Controller module Send_Preempt is called to actually issue the preempt.

Advance next readies the calling process and calls Scheduler. Upon the return from the call to Scheduler the Active Process Table is unlocked.

c. Ticket

Ticket returns a unique sequencer value with every invokation. The value returned will always be one more than the last value returned.

It is called with one argument, (a pointer to) the sequencer name. When invoked, Ticket asserts the global lock on the Active Process Table, effectively locking the Sequencer Table. Ticket then calls Locate_Seq with the pointer to the sequencer name given to it as an input argument and gets back the index of the sequencer in the Sequencer Table. It then obtains the sequencer's value which is to be returned to the calling module in the AX (Accumulator) Register following standard PL/M conventions. Before returning, ticket increments the value of the sequencer and unlocks the Active Process Table.

Note that Ticket does not call Scheduler like the other synchronization primitives Advance and Await. Ticket returns immediately from a call.

d. Read

Read returns the current value of an eventcount. It is called with one argument, (a pointer to) the name of the eventcount.

When called, Read locks the Active Process Table, so as to lock the Eventcount Table. It then calls Locate_Evc to obtain the index of the eventcount in the Eventcount Table. With this index, Read obtains the value of the eventcount and returns the value in the AX (Accumulator) Register following normal PL/M conventions. Prior to returning, Read unlocks the Active Process Table.

e. Tc_Pe_Handler

This module serves as the virtual preempt interrupt entry point into Scheduler. It is called by the Inner Traffic controller's Vp_Scheduler in the course of handling preempt interrupts.

Tc_Pe_Handler calls Scheduler to find the highest priority ready process to bind to the pre-empted virtual processor.

f. Create_Evc

This module creates an eventcount for a user process. Create_Evc is called with one argument, (a pointer to) the name of the eventcount to be created.

Upon being invoked, Create_Evc locks the Active Process Table, which effectively locks the Eventcount Table. It then calls Locate_Evc to determine whether or not the eventcount had already been created. This is to avoid making duplicate entries (since each process which will use the eventcount must declare at least the name). If the eventcount had not previously been created (viz., no entry is found in the Eventcount Table with the same name as given in the input argument) then an entry is made in the Eventcount Table. The name is copied into the Eventcount Table and the eventcount's current value is initialized to 0. Otherwise, no entry is made. Lastly, it unlocks the Active Process Table prior to returning.

g. Create_Seq

This module creates a sequencer for a user process. Create_Seq performs in exactly the same way as Create_Evc (paragraph f) except that it creates sequencers rather than eventcounts.

h. Create_Process

Create_Process provides the capability to dynamically create processes. It is called with one argument, a pointer to a process parameter block containing all the information necessary to initialize the process's stack and enter the newly created process into the Active Process Table. All of the process's segments had previously been loaded into memory by the system loader, Ross [16].

Create_Process first locks the Active Process Table. It then creates the initializaton stack frame. The process parameter block contains all of the initial register values (viz., initial values for all of the 8086's registers) for the process. These are stored in the initialization stack frame; the location of the stack is specified in the Process Parameter Block. The next step is to create the Active Process Table entry for the process. The affinity, priority and Stack Segment (SS) Register value are then entered in the Active Process Table. Lastly, Create_Process determines where this process should be inserted into the Load List based on its priority. Create_Process inserts the process into the Load List (viz., sets the Load Thread in the Active Process Table) immediately ahead of the first process it finds in searching down the Load$List whose priority is less than or equal to the newly created process. Finally, the Active Process Table is unlocked and execution returns to the caller. Note that the Scheduler is not called.

I. THE SUPERVISOR

In a general-purpose operating system the Supervisor provides common services such as library routines, linkers, various development tools and a file sytem. It also acts as the interface between user programs and the kernel.

1. General Description

At this state of the design, only one module resides at this level, a higher order language interface to the operating system kernel. This module (called the Gate) is constructed such that it is the only operating system module that the user must link to his processes to access kernel functions.

The Gate contains the actual linkages (viz., global procedure declarations) for all of the kernel functions. This allows the user to directly call on various kernel services without using absolute addresses that can change as the kernel continues to be developed. This structure allows the users and the operating system developers to continue their work independently without requiring the users to continually change their programs to accommodate changes in the kernel.

2. Supervisor Invocation (The Gate)

The Gate is actually a set of global (viz., PL/M PUBLIC) procedure declarations which the user programs can call directly. Each of the user accessible kernel functions is represented by one of these "procedures". In reality, they simply set up the required parameters and use a trap feature to effect the call to the "real" procedure of the same name residing in the kernel.

The Gate is written in assembly language because of the stack manipulation that must be done to enable the trap handler to 1) determine the correct kernel procedure to call, and 2) properly pass parameters to the kernel procedures. The trap handler in the kernel is an assembly language module as well. If the trap handler were written in PL/M, parameters would have to be somehow given to it explicitly prior to its calling on the kernel procedure. Since the trap handler is reached by an interrupt rather than a call, this is not possible. Instead, the parameters are moved on the stack to a position where they become parameters for the call by the trap handler to the kernel procedure.

This has the effect of de-coupling the user from all of the operating system modules below the level of the Supervisor.

V. CONCLUSIONS

A. SUMMARY OF RESULTS

The principal goal of this effort is the development of a multiple processor system. A parallel development effort in secure systems, Reitz [18], utilized the O'Connell and Richardson design as the basis for the kernel of a secure computer system utilizing the Zilog Z8000 microprocessor. The detailed designs of the kernels of both of these systems is nearly identical, at least at the level of kernel module interfaces. In both development efforts, no conceptual problems were encountered. Thus the O'Connell and Richardson design has been found to be consistent for multiple processors and secure computer systems.

System initialization [16], introduced a number of design changes. However, these had no adverse effect on the design or the system. Their integration is not a simple matter as they impact on the stack format, and the design of the process scheduler and virtual processor scheduler in that the accommodation of preempt interrupts is somewhat more difficult.

Another of the objectives is to test the viability of utilizing general-purpose, commercial microcomputer systems as the basic building blocks of multiple computer systems. It has been found that sufficiently developed microcomputer systems are available in industry. Further, it was determined that enough hardware support (busses, I/O devices, peripherals) is available to construct multiple computer systems without major hardware development efforts.

The state of the art in microcomputer software development was found to be less amenable. Such useful tools as high level languages, assemblers, etc. are available but they are generally limited to use with uniprocessor developmental systems. Additionally, most commercially available software development tools are highly machine dependent. Specifically, they require low-level monitors or special hardware that are only available on a development system. Thus there is little hope of easily modifying these tools to run on a different system than was intended by the vendor, particularly since details of their structure and operation are proprietary.

A. FURTHER RESEARCH

Further development work is still required. This includes the final construction of the Gate and the inclusion of two non-distributed kernel processes for I/O and memory management. These kernel processes provide for the virtualization of memory and I/O resources with which to achieve the goal of configuration independence.

The present design utilizes the test-and-set semaphore operation to implement global locks on kernel data bases (viz., the Active Process Table and the Virtual Processor Map). This mechanism (supported by the PL/M built-in procedure "Iockset" [5]) is a spin-lock with potentially significant impact on system bus traffic. This mechanism should be replaced by the Inner Traffic Controller synchronization primitives wherever possible to avoid the overhead of "busy-waiting".

This detailed design is considered to be only a first step in the development of a general-purpose multiple microcomputer system. O'Connell and Richardson's design offers some exciting opportunities to pursue development efforts in the areas of secure computer systems and fault tolerance.

APPENDIX A - PROGRAMMING

A. INTRODUCTION

This appendix is designed to be a practical introduction to programming methodology for this system.

Because there are multiple processors, a number of concepts and methodologies will necessarily be introduced which may at first be uncomfortable. This is especially true if one is firmly entrenched in the traditional concepts of the monolithic, sequential program structure. However, as one makes the transition to the concepts of process structuring, it will be seen to be a natural approach to the development of complex software systems. Additionally, it is essential to the effective use of multiple processor systems.

Parallelism immediately presents the programmer with an entirely new set of complexities. He is not limited to the strictly sequential execution of program statements in a single program. Exercising control over the order and timing of execution of multiple processes becomes a major part of the programming effort. Inter-process synchronization, the mechanism by which processes are controlled, is the most difficult concept which the user will be required to deal with. However, the synchronization primitives built into the operating system are designed to make this as simple and straightforward as possible.

It is assumed that the primary programming language for this system will be Intel's PL/M-86 [5, 10]. This is a powerful, block structured high level language designed for systems programming. This appendix is written assuming that the reader will program in PL/M-86 and is familiar with its terminology and notation. All of the examples make use of an informal PL/M notation.

B. THE PROCESS STRUCTURE

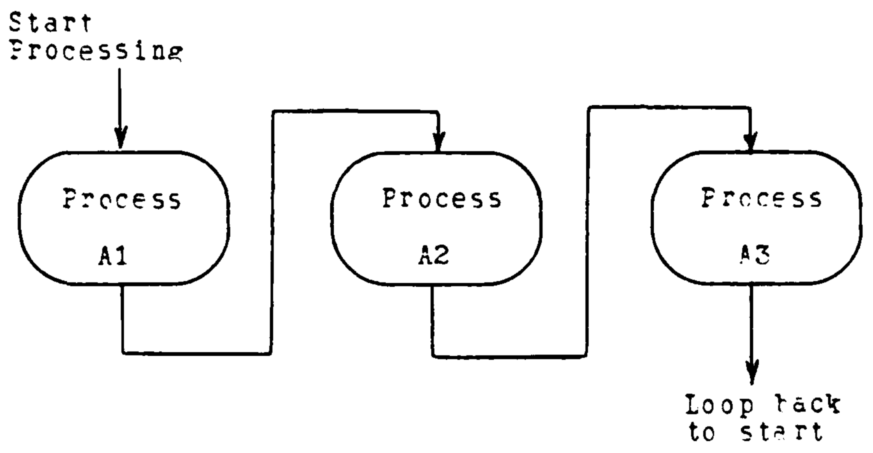
Consider the rather typical PL/M program module of Figure 9. It contains three procedure declarations and some mainline statements. Each of the procedures will execute when called from the mainline and, upon completion, will return control back to the mainline.

A single program is what most users are familiar with and is a structure which can be dealt with easily. However, as the computing task grows to any real size and complexity, this single program grows equally large and complex. The result is a huge program with a myriad of procedures that can only be called sequentially to perform necessary functions. Thus this structure does not allow taking advantage of the performance gain that parallel processing can offer.

```
Program Module A: Do;

      A1:  PROCEDURE /* Declaration */;
        DO;
         .
         .
         .
        END;
      END; /* Procedure A1 */

      A2: PROCEDURE; /* Declaration */
        DO;
         .
         .
         .
        END;
      END; /* Procedure A2 */

      A3: PROCEDURE; /* Declaration */
        DO;
         .
         .
         .
        END;
      END; /* Procedure A3 */

    DO; /* Begin Mainline */

      CALL A1;
      CALL A2;
      CALL A3;

    END; /* Mainline */

END; /* Program Module A */
```

EXAMPLE PL/M-86 PROGRAM

Figure 9



THREE PROCESSES EXECUTING SEQUENTIALLY

Figure 10

```
DECLARE NAME1(6) BYTE DATA ('NAME1%');

   Byte array of       String constant
   length 6 to hold    name defined by
   the name            the user
```

DECLARATION OF EVENTCOUNT AND SEQUENCER NAMES

Figure 11

The principal advantages of the process structure lie in the ability to utilize multiple processes and to independently construct individual components of software subsystems, viz., processes. Rather than using a single process to accomplish the entire job as in Figure 9, the overall task can be partitioned and accomplished by a number of smaller cooperating processes. Each of these processes can be smaller than the single monolithic program and so is easier to design, implement and test. This allows entire processes (each a distinct program) to be developed and tested semi-independently in a manner similar to the development and testing of individual procedures in a single PL/M program.

Control over processing functions is also much more flexible. One is not forced into a strictly sequential series of procedure calls. Many processes can be allowed to execute in parallel, which can bring about dramatic gains in overall performance.

Figure 10 is a simple example of the flow of execution in a system with three processes. The three processes perform exactly the same functions as the three procedures of Figure 9 and so bear the same names. In this example the processes execute sequentially, one after the other in a set order. Processing goes on forever in this "loop". Process A2 will only begin executing after it has been somehow "signalled" by process A1. The same is true of process A3 whose execution is synchronized with process A2. Obviously, there must be some control mechanism that allows these processes to do this.

C. INTER-PROCESS SYNCHRONIZATION MECHANISMS

The ability to synchronize the execution of processes throughout the system, (irrespective of which microcomputer they are loaded on), is the cornerstone of the power and flexibility of this system. To accomplish this, process synchronization is based on the notion of events.

1. Events

An event is anything that one considers significant and can direct, in some fashion, the computer to respond to. As an example, consider a clock which indicates a time of twelve o'clock. The computer has no inherent conception of time. As far as it is concerned, time may be nothing more than a value in some register. In some way, then, time must be defined for the computer. This is accomplished by translating the occurrence of twelve o'clock into an event. When the event occurs, the computer recognizes that it is to respond in some specified manner.

Events are defined so as to be very general in nature. They can be used to represent the completion of a program, as in the completion of process A1 in Figure 10 which started the execution of process A2. They can represent virtually anything of interest to the programmer, at least anything that he can identify as being of significance.

2. Eventcounts and Sequencers

Eventcounts and sequencers allow processes to synchronize with each other somewhat indirectly. To synchronize directly, a process would have to somehow identify the other processes with which it is synchronizing (viz., explicitly signal a process by name). This would require the naming of individual processes or some similar identification scheme.

Rather than using a process naming scheme, the individual processes "agree", in a sense, to cooperate by using a common set of memory objects called eventcounts and sequencers. In this way, even though the processes must know the names of the eventcounts and sequencers that they use, they are not required to know anything at all about each other's identities. In fact, a process need not even know how many other processes will tbe synchronizing with it. This offers some advantages in parallel processing. Processes that synchronize with eventcounts do not have to know how many other processes will also use the same eventcounts. This means that fewer coding changes will be required when, for example, a single process is partitioned into several processes all executing in parallel. All of the "new" processes will synchronize on the same eventcount so that no changes are required in the process that originally synchronized with the single process.

Eventcounts are used to keep track of the occurrence of specific events. They are managed for the user by the system. Eventcounts are implemented as PL/M-86 word variables ranging in value from 0 to 65536. Sequencers are also implemented as PL/M-86 word variables ranging in value from 0 to 65536. However, sequencers can be used to impose an order on the occurrence of events. They are thus used with eventcounts to provide for mutual exclusion.

3. Eventcount and Sequencer Declarations

a. Declaring Eventcount and Sequencer Names

Eventcounts and sequencers are named using a byte array of alphanumeric characters. The format for declaring an eventcount or sequencer name is given in Figure 11. Note that the names are constants, not variables. Once declared, a name must not change. Eventcount and sequencer names consist of 5 characters followed by a per cent symbol (%). Note in Figure 11 that the name of the byte array must be the same as the string constant given in the DATA Initialization. This allows the user to reference the eventcount or sequencer by name and allows the operating system to identify it.

Remember that the names of eventcounts and sequencers must be declared in exactly the same way in each PL/M-86 module in which they will be used.

b. Passing Eventcount and Sequencer Names

When calling the operating system synchronization primitives, eventcount and sequencer names are always passed as PL/M-86 location references using the "@" operator. As an example, consider that a byte array called "NAME1" holds the string "NAME1%" (note that the "%" symbol is only a delimiter and is not considered to be part of the name). To pass the name in a call to an operating system synchronization primitive, then, the parameter "@NAME1" is used. With the pointer so given, the operating system can "read" the name directly from the array.

c. Creating Eventcounts and Sequencers

Before an eventcount or a sequencer is used, the operating system must be informed of its existence. This is accomplished by a call to the operating system procedures CREATE$EVC (for eventcounts) and CREATE$SEQ (for sequencers). The format of these operations is shown in Figures 12 and 13. There is only one argument for either of the calls, the pointer to the previously declared name. When created, an eventcount or a sequencer will always be initialized with a starting value of 0.

4. Synchronization

Eventcounts and sequencers are utilized by means of a set of operations which may be performed on them. The user cannot directly perform operations on either eventcounts or

```
CALL     CREATE$EVC(@NAME1);

         Kernel      Pointer to
         function    the head of the
         name        pre-declared
                     byte array holding
                     the string name
```

CREATING AN EVENTCOUNT

Figure 12

```
CALL     CREATE$SEQ(@NAME1);

         Kernel      Pointer to
         function    the head of the
         name        pre-declared
                     byte array holding
                     the string name
```

CREATING A SEQUENCER

Figure 13

sequencers, but rather calls on certain operating system primitives to do these for him.

a. Operations on Eventcounts

There are three operations that one can perform on eventcounts. They are ADVANCE, AWAIT and READ. ADVANCE and AWAIT are untyped procedures. READ is a value returning typed procedure (function call) that returns a PL/M-86 word value to the calling process.

An example of a READ operation is given in Figure 14. The READ operation allows the user to obtain the current value of a specified eventcount. READ returns the eventcount's value in the AX Register (in accordance with normal PL/M-86 conventions). Thus a process calls READ with the name of the eventcount as the argument and gets back the eventcount's current value. Note in Figure 14 that the current value of eventcount EVENT is returned to the user-defined word variable "WORD$VARIABLE".

The AWAIT operation, Figure 15, is used by a process to block itself (suspend its execution) until the eventcount reaches the value specified in the call. AWAIT requires two arguments, the eventcount name and the event (actually the value of the eventcount) to wait for. The value for which the process will wait must be a PL/M-86 word value. This allows the process to synchronize itself with

```
WORD$VARIABLE = READ(@EVENT);
```

THE READ OPERATION

Figure 14

```
CALL AWAIT(@EVENT,VALUE$TO$AWAIT);
```

THE AWAIT OPERATION

Figure 15

```
CALL ADVANCE(@EVENT);
```

THE ADVANCE OPERATION

Figure 16

```
WORD$VARIABLE = TICKET(@NAME1);
```

THE TICKET OPERATION

Figure 17

other processes by waiting, for instance, until a set of data is ready for it to use.

The ADVANCE operation, Figure 16, is used to signal the occurrence of an event. ADVANCE only requires one argument, the name of the eventcount to be advanced. When it is called, it will cause the value of the eventcount to be incremented by one. The operating system will then proceed to unblock those processes that were waiting for the eventcount to reach the current value (by virtue of having previously called AWAIT).

b. Operations on Sequencers

There is only one operation that can be performed on sequencers. It is called TICKET, Figure 17. TICKET is a value returning typed procedure (function call) similar to the READ operation for eventcounts. However, TICKET returns to the caller a unique sequencer value. The current value of the sequencer is returned to the caller and then the sequencer is incremented by one for the next caller time a TICKET operation is performed on it. This will be true irrespective of how many different processes perform the TICKET operations. In this way TICKET provides the totally ordered set of values for use by multiple processes in effecting mutual exclusion.

5. Synchronization Examples

a. Sequential Processing Example

Figure 18 provides a detailed example of how a process would be programmed to actually create and use eventcounts for synchronization. The program shown here is actually process A1 of Figure 17.

Referring to the flow of control in Figure 17, it can be seen that process A2 will begin execution when signalled by A1. Similarly process A3 will begin when signalled by A2. Finally, when A3 signals its completion, the "loop" starts over again with process A1. This is reflected in the sample program for process A1, Figure 18. Here two eventcounts are declared and created, "ENDA1" and "ENDA3". Eventcount ENDA1 is used to synchronize with process A2. Specifically, ENDA1 refers to the event corresponding to the completion of A1's processing task. The occurrence of this event is signalled to process A2 through the Advance operation performed on eventcount ENDA1 (located at the end of the "Do Forever" loop). The result of the Advance is to start the execution of process A2. After the call to Advance, process A1 will loop back to the call to Await with an awaited value of 1 this time and (if process A3 has not yet advanced ENDA3) will wait there.

Process A2 is programmed as shown in Figure 19. Note that it first calls Await with the eventcount ENDA1 and an awaited value of 1. This is in contrast to the awaited

```
PROGRAM MODULE A1: DO;

 /* Declare Eventcounts */
 DECLARE ENDA1(6) BYTE DATA ('ENDA1%');
 DECLARE ENDA3(6) BYTE DATA ('ENDA3%');

 /* Declare a local word variable */
 DECLARE A1$AGAIN WORD;

 /* Declare Synchronization Primitives */
 CREATE$EVC: PROCEDURE(EVENTCOUNT) EXTERNAL;
             DECLARE EVENTCOUNT POINTER;
 END;
 AWAIT: PROCEDURE(EVENTCOUNT, VALUE) EXTERNAL;
        DECLARE EVENTCOUNT  POINTER,
                VALUE       WORD;
 END;
 ADVANCE: PROCEDURE(EVENTCOUNT) EXTERNAL;
          DECLARE EVENTCOUNT POINTER;
 END;

     /* Begin Mainline */

A1$AGAIN = 0; /* To start execution immediately */
CALL CREATE$EVC(@ENDA1); /*
CALL CREATE$EVC(@ENDA3);

DO WHILE 1; /* Do Forever */
  /* Check to see if processing should begin */
  CALL AWAIT(@ENDA3,A1$AGAIN);
           .
           .
           .
  /* Processing completed so notify process A2 */
  CALL ADVANCE(@ENDA1);
  /* Increment the value to await */
  A1$AGAIN = A1$AGAIN + 1;

END; /* Of Do Forever */

END; /* Module */
```

EXAMPLE CODE FOR PROGRAM A1

Figure 18

```
PROGRAM MODULE A2: DO;

 /* Declare Eventcounts */
 DECLARE ENDA1(6) BYTE DATA ('ENDA1%');
 DECLARE ENDA2(6) BYTE DATA ('ENDA2%');

 /* Declare a local word variable */
 DECLARE A2$AGAIN WORD;

 /* Declare Synchronization Primitives */
 CREATE$EVC: PROCEDURE(EVENTCOUNT) EXTERNAL;
               DECLARE EVENTCOUNT POINTER;
 END;
 AWAIT: PROCEDURE(EVENTCOUNT,VALUE) EXTERNAL;
        DECLARE EVENTCOUNT  POINTER,
                VALUE       WORD;
 END;
 ADVANCE: PROCEDURE(EVENTCOUNT) EXTERNAL;
          DECLARE EVENTCOUNT POINTER;
 END;

     /* Begin Mainline */

     A2$AGAIN = 1; /* To start execution after process A1 */
     CALL CREATE$EVC(@ENDA1); /*
     CALL CREATE$EVC(@ENDA2);

DO WHILE 1; /* Do Forever */
   /* Check to see if processing should begin */
   CALL AWAIT(@ENDA1,A2$AGAIN);
              .
              .
              .
  /* Processing completed so notify process A3 */
  CALL ADVANCE(@ENDA2);
  /* Increment the value to await */
  A2$AGAIN = A2$AGAIN + 1;

END; /* Of Do Forever */

END; /* Module */
```

EXAMPLE CODE FOR PROGRAM A2

Figure 19

```
PROGRAM MODULE A3: DO;

 /* Declare Eventcounts */
 DECLARE ENDA2(6) BYTE DATA ('ENDA2%');
 DECLARE ENDA3(6) BYTE DATA ('ENDA3%');

 /* Declare a local word variable */
 DECLARE A3$AGAIN WORD;

 /* Declare Synchronization Primitives */
 CREATE$EVC: PROCEDURE(EVENTCOUNT) EXTERNAL;
             DECLARE EVENTCOUNT POINTER;
 END;
 AWAIT: PROCEDURE(EVENTCOUNT,VALUE) EXTERNAL;
        DECLARE EVENTCOUNT  POINTER,
                VALUE       WORD;
 END;
 ADVANCE: PROCEDURE(EVENTCOUNT) EXTERNAL;
          DECLARE EVENTCOUNT POINTER;
 END;

     /* Begin Mainline */

A3$AGAIN = 1; /* To start execution after process A2 */
CALL CREATE$EVC(@ENDA2); /*
CALL CREATE$EVC(@ENDA3);

DO WHILE 1; /* Do Forever */
  /* Check to see if processing should begin */
  CALL AWAIT(@ENDA2,A3$AGAIN);
         .
         .
         .
  /* Processing completed so notify process A1 */
  CALL ADVANCE(@ENDA3);
  /* Increment the value to await */
  A3$AGAIN = A3$AGAIN + 1;

END; /* Of Do Forever */

END; /* Module */
```

EXAMPLE CODE FOR PROGRAM A3

Figure 20

value of Ø used by process A1 in its initial call to Await. Thus process A2 will wait at this point until signalled by process A1 (if process A2 begins executing before process A1). After A1 performs an Advance on eventcount ENDA1, A2 will perform its processing and when complete will signal process A3 to begin via an Advance operation on the eventcount ENDA3. As with process A1, it will then loop back to the Await operation and will be suspended until A1 once again signals it to continue.

Figure 20 shows the program for process A3. Process A3 performs an initial Await as the others did and when its processing task has been completed, it signals process A1 to begin the "loop" again via an Advance operation on eventcount ENDA3.

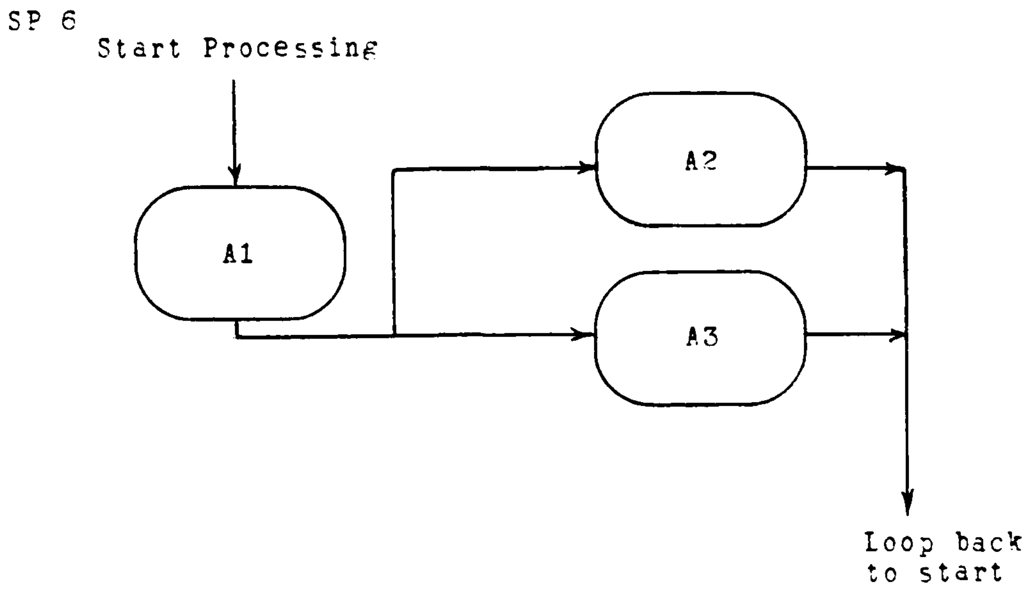
These three processes are intended to demonstrate the mechanics of synchronizing with eventcounts. As can be seen, the operations used in all three of the processes are very similar. The real differences lie only in the specific eventcounts that each process uses in the calls to Await and Advance. Note, however that each process performs the Await operation at a point that ensures the process will be synchronized with its companion processes even if the process begins "out of order". This is required to avoid confusion since there is no guarantee that the first of the three processes to begin executing will be the one intended by the programmer to execute first (viz., A1 in this example).

b. Parallel Processing Example

Suppose that instead of the simple sequential execution of processes, as in the above example, one wishes to execute processes in parallel. The eventcount mechanism provides the capability to synchronize parallel processes in (mechanically) the same way that sequential processing is accomplished.

Consider again the three processes A1, A2, and A3 from the previous example. This time the programmer notes that processes A2 and A3 both depend on input data (a set of filter coefficients, for example) from process A1. However, he also notes that neither process A2 nor A3 alters the input data (they only read it). Thus processes A2 and A3 become candidates for parallel execution since they both have a common event upon which to begin execution (the point where the input data becomes available) and they do not depend on each other. Note, however, they must reside in different microcomputers for their execution to actually occur in parallel.

The desired flow of execution is shown in Figure 21. Implementing the parallel execution of processes A2 and A3 is actually a simple task. Only process A3 need be changed. Processes A2 and A3 await the same value of a common eventcount rather than different ones. Thus the



FLOW OF CONTROL IN PARALLEL PROCESSING

Figure 21

```
PROGRAM MODULE A3: DO;

   /* Declare eventcounts */
   /* Eventcounts ENDA1, ENDA2 and ENDA3     */

   /* Declare a local word variable */
   DECLARE A3$AGAIN WORD;

   DECLARE ENDA1(6) BYTE DATA ('ENDA1%');
   DECLARE ENDA2(6) BYTE DATA ('ENDA2%');
   DECLARE ENDA3(6) BYTE DATA ('ENDA3%');

   /* Declare synchronization primitives */
   /*          Advance and Await           */

            /* Begin Mainline */

       /* Create the eventcounts */

       A3$AGAIN = 1;

       DO WHILE 1; /* Do forever */

           CALL AWAIT(@ENDA1,A3$AGAIN);
                    .
           /*   Perform processing   */
                    .
           /* Processing of both A2 and A3 complete */
           CALL AWAIT(@ENDA2,A3$AGAIN);
           CALL ADVANCE(@ENDA3);

           /* Increment the value to await */
           A3$AGAIN = A3$AGAIN + 1;

       END; /* Of Do Forever */

END; /* Of Module */
```

PARALLEL PROCESSING EXAMPLE PROCESS A3

Figure 22

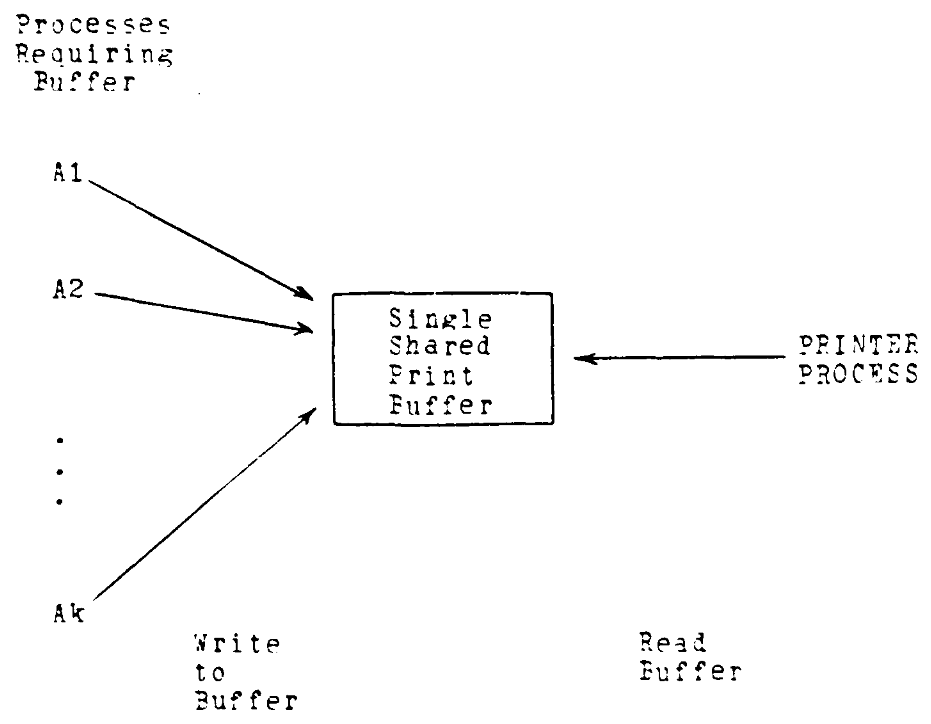
result of a single Advance on the eventcount will be to simultaneously signal processes A2 and A3.

The operations performed by process A3 is shown in Figure 22. Process A1 is still required to perform its processing first to provide input data for processes A2 and A3. Thus process A1 performs an initial Await operation on the eventcount ENDA3 with an awaited value of 0, and since the eventcount is initialized to a value of 0 upon creation, A1 will be allowed to continue. Processes A2 and A3 both perform their initial Await operations on the eventcount ENDA1 using the same awaited value (they each wish to begin processing when the set of input data becomes available). However, process A3 will advance the value of ENDA3 only after both A2 and A3 have completed. This allows A1 to wait for the two events to occur (viz., the completion of processes A2 and A3) before it begins again. Thus with the single Advance operation performed by A1 on ENDA1 two processes begin execution.

This example has shown how the programmer can easily make use of eventcounts to synchronize parallel processes with the same methodology used for sequential processes.

c. Mutual Exclusion Example

Mutual exclusion is required in certain situations where one and only one process can be allowed to access a shared resource (some set of data) at a time.



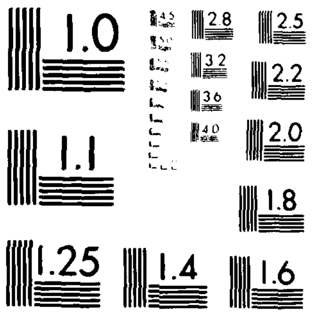
MUTUAL EXCLUSION EXAMPLE

Figure 23

```
PROGRAM MODULE PRINTER PROCESS: DO;

  /* Declare eventcounts FULLB and EMPTY, used */
  /* by all of the processes                   */
  DECLARE FULLB(6) BYTE DATA ('FULLB%');
  DECLARE EMPTY(6) BYTE DATA ('EMPTY%');

  /* Declare a local word variable */
  DECLARE AGAIN WORD;

  /* Declare synchronization primitives */
  /*         Advance and Await          */

        /* Begin Mainline */

     /* Create the eventcounts */

     DO WHILE 1; /* Do forever */

        CALL AWAIT(@FULLB,AGAIN);
                  .
        /*   Perform processing   */
                  .
        /* Processing complete, notify others */
        /* that buffer is available           */
        CALL ADVANCE(@EMPTY);

        /* Increment the value to await */
        AGAIN = AGAIN + 1;

     END; /* Of Do Forever */

END; /* Of Module */
```

PRINTER PROCESS FOR MUTUAL EXCLUSION EXAMPLE

Figure 24


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

```
PROGRAM MODULES A1 THROUGH Ak: DO;

  /* Declare eventcounts FULLB and a sequencer */
  /* used by all of the processes.  Sequencer  */
  /* by all of the processes.  Sequencer is    */
  DECLARE FULLB(6) BYTE DATA ('FULLB%');
  DECLARE EMPTY(6) BYTE DATA ('EMPTY%');
  DECLARE TURNA(6) BYTE DATA ('TURNA%');

  /* Declare some local variables among which is T */
  DECLARE T WORD;

  /* Declare synchronization primitives       */
  /* Advance and Await as before plus Ticket */
  TICKET: PROCEDURE(SEQUENCER) WORD EXTERNAL;
          DECLARE SEQUENCER POINTER;
  END;

        /* Begin Mainline */
               .
               .
               .
     /* At this point process needs to print data      */
     /* so create the eventcounts as usual and create */
     /* the sequencer                                  */
     CALL CREATE$SEQ(@TURNA);

     /* Obtain a "turn" for the buffer */
     T = TICKET(@TURNA);

     /* Wait for "turn" to come up *
     CALL AWAIT(@EMPTY,T);

     /* When awakened, process may use the */
     /* buffer (its "turn" has come up)    */

     /* Finished with the buffer so notify   */
     /* Printer Process that there is output */
     CALL ADVANCE(@FULLB);

 END; /* Of Module */
```

PROCESSES A1 - Ak FOR MUTUAL EXCLUSION EXAMPLE

Figure 25

Sequencers are used in conjunction with eventcounts to solve these types of problems.

To illustrate mutual exclusion, consider the flow of control in Figure 23. Here an unknown number of processes (A1 through Ak) require access to a single resource (used by A3). Printer Process is some printer service and the single shared resource is its input buffer. Obviously only one of the processes requesting printer services can be allowed to write into the input buffer at a time and no process can write into the buffer while Printer Process is trying to transfer the information to the printer.

The solution is shown in Figures 24 and 25. Figure 24 shows the programming of Printer Process and Figure 25 shows how each of the processes requiring printer services are programmed.

In Figure 16, Printer Process only requires the use of eventcounts (since it does not alter the data in the input buffer). It only needs to know when to begin transferring the data and to signal that the buffer is free upon completion of the transfer. Thus Printer Process only uses two eventcounts, FULL and EMPTY corresponding to the buffer's containing data from a process (FULL) and its being emptied by Printer Process (EMPTY). Thus Printer Process performs an Await operation on FULL and waits for an input process to give it some data. When a process performs an

Advance on FULL, Printer Process will be awakened to output it. When Printer Process finishes outputting the data, it will perform an Advance on the eventcount EMPTY and loop back to the AWAIT.

The other processes, Figure 25, are to use the same eventcounts, performing Awaits on the eventcounts EMPTY (waiting for the buffer to become available) and FULL (signalling Printer Process that there is data to print out). However, the awaited value is derived from a TICKET operation on the sequencer TURN. Note that each of these processes will perform TICKET operations on the same sequencer (TURN) and so will all receive unique awaited values ("turns", as in taking a number from a ticket machine at a department store, [13]) for the buffer. These TICKET operations will return a unique value for the sequencer every time it is called irrespective of which process calls TICKET (provided the same sequencer is used as the argument). Then the processes simply wait for their "turns" to come up. Since each process will wait for its "turn", there will only be one process writing into the buffer at a time.

This example demonstrates the use of sequencers in mutual exclusion problems. As can be seen, the use of sequencers provides a very simple way to mediate access to shared resources, particularly useful when the number of processes involved is not known in advance or may change.

D. THE OPERATING SYSTEM GATE

Somehow there must exist a linkage between the user processes and the operating system to use the functions outlined in the preceeding paragraphs. This is provided by linking to each user process one operating system module known as a GATE. The GATE contains the Public declarations for the synchronization procedures which the user may access. The GATE, then, allows the user to call operating system procedures in exactly the same way that any EXTERNAL procedure would be called. The advantage is that only the GATE (which is very small) must be linked and loaded with each user process, not the entire operating system. Additionally, during system generation [16], the Gate is located (PL/M terminology for the assignment of absolute addresses) in exactly the same place in memory for all of the processes. The result is that the Gate segments loaded in with each process will be overlayed. Thus all of the processes on a single microcomputer will share the same copy of the Gate code. This minimizes the amount of physical memory used by the Gate.

Figure 26 tabulates the required format for all of the external procedure declarations that must be included in each user module making use of operating system functions. Note that only the functions actually used need to be declared.

Creating an Eventcount:

```
CREATE$EVC: PROCEDURE(EVENTCOUNT) EXTERNAL;
            DECLARE EVENTCOUNT POINTER;
END;
```

Creating a Sequencer:

```
CREATE$SEQ: PROCEDURE(SEQUENCER) EXTERNAL;
            DECLARE SEQUENCER POINTER;
END;
```

The Advance Operation:

```
ADVANCE: PROCEDURE(EVENTCOUNT) EXTERNAL;
         DECLARE EVENTCOUNT POINTER;
END;
```

The Await Operation:

```
AWAIT: PROCEDURE(EVENTCOUNT,VALUE) EXTERNAL;
       DECLARE EVENTCOUNT  POINTER,
               VALUE WORD;
       END;
```

The Ticket Operation:

```
    TICKET: PROCEDURE(SEQUENCER) BYTE EXTERNAL;
            DECLARE SEQUENCER POINTER;
    END;
```

The Read Operation:

```
    READ: PROCEDURE(EVENTCOUNT) BYTE EXTERNAL;
          DECLARE EVENTCOUNT POINTER;
    END;
```

KERNEL CALL EXTERNAL PROCEDURE DECLARATIONS

Figure 26

E. SHARED PROGRAM CODE

Processes can be made to share code as long as they are all loaded on the same microcomputer and the shared modules all have the 'REENTRANT' attribute. This places all variable storage on the stack so that there is no confusion when two processes try to invoke the module at the same time.

Because the system is bus-oriented (all of the microcomputers share a single system bus), code sharing should not usually be forced for processes which reside in different microcomputers. This requires access to the system bus for instruction fetches making this technique less efficient. Therefore, global sharing of code is not not the expected convention during system generation, although it is not prevented outright [16]. In fact the programmer will not be in direct command as the system generation operator will make this decision.

One rule of thumb that quite often applies to attempts at optimization is that the memory that is saved is paid for with a loss of speed. Quite often one can speed execution up drastically if he is not overly concerned about using memory.

In summary, the sharing of code segments to save memory is a technique that is discouraged in this system if the processes which share them reside on different microcomputers. It will "work", of course, but has a very detrimental effect on performance.

F. SHARED DATA

Sharing of data between processes is tightly-coupled in that the data is not explicitly transmitted from one process to another. Rather, data sharing is accomplished by using shared PL/M data segments. These shared data segments can reside in global memory where they are directly accessible to the processes concerned.

PL/M allows one to develop programs modularly by providing data declarations with PUBLIC and EXTERNAL attributes. When the modules are linked, all of the declared variables (such as byte, word and pointer quantities, arrays, structures, etc.) are collected into a single data segment for the program. Thus PL/M-86 expects that each program will have its own local data segment.

In modules where a variable is declared with the EXTERNAL attribute, it is understood that the variable may actually reside in a non-local data segment. The intention is that eventually, when all of these modules are linked together into one program the PUBLIC and EXTERNAL references will be resolved.

Processes, though are not linked together. They are altogether independent PL/M programs. However, one can share data in much the same way as the modules in a single PL/M program by declaring all shared data in the processes with the EXTERNAL attribute. Thus each process will be aware of the existence of a separate data segment. The shared data segment is then separately created as a PL/M module containing only shared data declared with the PUBLIC attribute -- no local data or code is ever included. This module is then compiled separately and linked to each of the processes sharing the data as if it were simply another program module. The only difference is that this module will only have a meaningful data segment. The code segment will be empty.

It must be emphasized that such data segments are the only means of communication between processes. In particular, a reference to an absolute address (including constant or computed pointer values) is NEVER allowed. To do so will destroy the integrity of this operating system design.

G. PRIVILEGED INSTRUCTIONS

Because the operating system controls the physical resources of the system, certain instructions which are valid in either the high level language PL/M or the 8086 assembly language ASM-86 may not be used. The reason for this is that their use will interfere with the correct operation of the system.

1. Interrupt Masking

The operating system uses the interrupt structure of the system for its own purposes. Because of this, the user must never, repeat NEVER, mask interrupts using the assembly language CLI/STI instructions or the PL/M-86 DISABLE/ENABLE instructions.

The operating system uses interrupts system-wide during normal operation and requires that interrupts be unmasked at all times while user processes are executing.

This is not to be confused with the use of interrupt handler routines which are required for certain software packages, notably the PL/M-86 real number library routines. These will not interfere with system operation.

2. Input/Output Operations

Direct access to Input/Output facilities is also the purview of the operating system. Thus the user is also prohibited from using the PL/M and ASM-86 I/O instructions. Instead, a system service is provided to perform I/O functions for the user.

APPENDIX B - KERNEL MODULES

This section contains the detail "pseudo-code"
for the kernel modules. These have not been
fully tested and should only be considered an
aid to understanding and not final code.

```
/*************************************************************/
/*              THE INNER TRAFFIC CONTROLLER                 */
/*************************************************************/


/*************************************************************/
/*              VIRTUAL PROCESSOR SCHEDULER                  */
/*************************************************************/

; External PL/M-86 procedures called by this module
           EXTRN GETWORK:   FAR,
                 RUNTHISVP: FAR,
                 RDYTHISVP: FAR,
                 LOCKVPM:   FAR,
                 UNLOCKVPM: FAR

SCHEDULER SEGMENT

  PUBLIC VPSCHEDULER

  VPSCHEDULER PROC FAR

           ASSUME CS:SCHEDULER
           ASSUME DS:NOTHING
           ASSUME SS:NOTHING
           ASSUME ES:NOTHING

; Entry point for a call to VpScheduler
; Establish activation record, save registers that
;   VpScheduler will use.

             PUSH DS
             PUSH AX
             PUSH CX
             PUSH BP
             CALL LOCKVPM
             CALL RDYTHISVP
             MOV BP,SP
```

```
              MOV CX,ØH    ; "ØH" indicates normal return

; Entry point for a preempt interrupt.  Reached by a jump
;   from ITC_PREEMPT_HANDLER procedure.

  INT_ENTRY:  PUSH CX
              CALL GETWORK ; Returns new "DBR" in the
                           ; AX register
              POP CX

; Swap virtual processors.  This is accomplished by saving
;   the SP and BP registers in known locations at the base
;   of the stack along with the VpScheduler return type
;   flag.  The process bound to the selected virtual
;   is accessible via the address space descriptor,
;   the SS register value.

              MOV SS:WORD PTR Ø,SP
              MOV SS:WORD PTR 2,BP
              MOV SS:WORD PTR 4,CX ; Return type flag
              MOV SS,AX            ; New address space desc.

; Swap is complete at this point since the SS register
;    now holds the new stack segment value

              MOV SP,SS:WORD PTR Ø
              MOV BP,SS:WORD PTR 2
              PUSH AX
              CALL RUNTHISVP
              MOV CX,SS:WORD PTR 4

;   Check VpScheduler return type flag to determine the type
;      of return required for the process.
              CMP CX,77H    ; Return type flat = Interrupt?
              JNE NORM_RET ; If not, do a normal return
              JMP INT_RET  ; If so, do an interrupt return
              NORM_RET:  CALL UNLOCKVPM
              POP BP
              POP CX
              POP AX
              POP DS
              RET

  VPSCHEDULER ENDP

  ITC_PREEMPT_HANDLER PROC FAR

             ASSUME CS:SCHEDULER
```

```
                ASSUME DS:NOTHING
                ASSUME SS:NOTHING
                ASSUME ES:NOTHING

                INT_VEC:  CLI
                PUSH ES
                PUSH DS
                PUSH AX
                PUSH CX
                PUSH DX
                PUSH BX
                PUSH SI
                PUSH DI
                CALL LOCKVPM
                CALL RDYTHISVP
                JMP INT_ENTRY
      INT_RET:  CALL UNLOCKVPM
                CALL CHECKPREEMPT
                POP DI
                POP SI
                POP BX
                POP DX
                POP CX
                POP AX
                POP DS
                POP ES
                IRET

  ITC_PREEMPT_HANDLER ENDP

 SCHEDULER ENDS

/***********************************************************/
/*     Virtual Processor Scheduler Internal Modules        */
/***********************************************************/

/*   Externally Defined Variable Declarations              */

DECLARE VPM(1) STRUCTURE
        (VP$STATE       BYTE,
         VP$PRIORITY    BYTE,
         EVC$AW$ID      BYTE,
         EVC$AW$VALUE   WORD,
         SS$REG         WORD,
         PE$PEND        BYTE) EXTERNAL;

DECLARE VPM$LOCK BYTE EXTERNAL;
```

```
DECLARE(CPU$NUMBER,VP$START,VP$END,VPS$PER$CPU)
                  BYTE EXTERNAL;

DECLARE IDLE$DBR WORD EXTERNAL;

DECLARE CPU$INT$VECTOR(16) BYTE EXTERNAL;

/*    Literal constants                                    */

DECLARE FALSE   LITERALLY   '0',
        READY   LITERALLY   '1',
        RUNNING LITERALLY   '3',
        WAITING LITERALLY   '7',
        IDLE    LITERALLY  '15',
        TRUE    LITERALLY '119';

/*    External Procedure Declarations                      */

TC$PE$HANDLER: PROCEDURE EXTERNAL;
END;


/**********************************************************/
/*     GETWORK Procedure                                  */
/*--------------------------------------------------------*/
/*     Function call.  Searches the Virtual Processor Map */
/*     the highest priority runnable virtual processor    */
/*     (state is either ready or idle with the Preempt    */
/*     Pending Flag set).  Returns the DBR value (SS      */
/*     Register value) of the bound process in the Ax     */
/*     Register.                                          */
/**********************************************************/

GETWORK: PROCEDURE WORD REENTRANT PUBLIC;

           DECLARE (PRI,I)      BYTE;
           DECLARE SELECTED$DBR WORD;

/* Begin search of the Virtual Processor Map using the     */
/* priorities.  Initially set to the lowest possible.      */
PRI = 255;

DO /* Search Virtual Processor Map for the highest       */
   /* priority ready virtual processor to run.         */
   I = VP$START TO VP$END;

   IF /* The virtual processor can be selected, it is  */
```

```
     /* is either the ready state, or the idle state    */
     /* with a virtual preempt pending.                  */
    ((VPM(I).VP$PRIORITY < PRI) AND
    (VPM(I).VP$STATE = READY OR
    (VPM(I).VP$STATE = IDLE AND
    VPM(I).PE$PEND = TRUE))) THEN

    DO; /* Select the virtual processor. */
      SELECTED$DBR = VPM(I).SS$REG;
      PRI = VPM(I).VP$PRIORITY;
    END; /* Do. Select the virtual processor. */

END; /* DO loop search of the Virtual Processor Map. */

/* Return the SS$REG value of the selected virtual          */
/* processor in the AX (Accumulator) Register.
RETURN SELECTED$DBR;

END; /* GETWORK Procedure. */

/*************************************************************/
/*      RUNTHISVP Procedure                                  */
/*-----------------------------------------------------------*/
/*      Sets the selected virtual processor to running.      */
/*      Searches the Virtual Processor Map with the          */
/*      process's SELECTED$DBR.                              */
/*************************************************************/

RUNTHISVP: PROCEDURE(VP$DBR) REENTRANT PUBLIC;
           DECLARE VP$DBR WORD,
                   VP     BYTE;

  VP = VP$END;

  DO WHILE /* Look for the VP with this SS$REG value. */
     (VPM(VP).SS$REG <> VP$DBR);

     VP = VP - 1;

  END; /* Do While */

  VPM(VP).VP$STATE = RUNNING;

  RETURN;
END; /* RUNTHISVP Procedure. */

/*************************************************************/
/*      RDYTHISVP Procedure                                  */
```

```
/*--------------------------------------------------------------*/
/*     Sets the currently running virtual processor's           */
/*     state to ready.                                          */
/****************************************************************/

RDYTHISVP: PROCEDURE REENTRANT PUBLIC;

     VPM(ITC$RET$VP).VP$STATE = READY;

     RETURN;
END; /* RDYTHISVP Procedure */

/****************************************************************/
/*     LOCKVPM Procedure                                        */
/*--------------------------------------------------------------*/
/*     Locks the Virtual Processor Map.                         */
/****************************************************************/

LOCKVPM: PROCEDURE REENTRANT PUBLIC;

     /* PL/M-86 built-in spin-lock procedure. */
     DO WHILE LOCKSET(@VPM$LOCK,119);
     END;

     RETURN;
END; /* LOCKVPM Procedure. */

/****************************************************************/
/*     UNLOCKVPM Procedure                                      */
/*--------------------------------------------------------------*/
/*     Unlocks the Virtual Processor Map.                       */
/****************************************************************/

UNLOCKVPM: PROCEDURE REENTRANT PUBLIC;

     VPM$LOCK = 0;

     RETURN;
END; /* UNLOCKVPM Procedure. */

/****************************************************************/
/*      HDWR$INT Procedure                                      */
/*--------------------------------------------------------------*/
/*      Generates a hardware preempt interrupt.                 */
/****************************************************************/

HDWR$INT: PROCEDURE(CPU) REENTRANT PUBLIC;
          DECLARE CPU   BYTE,
                    PORTA LITERALLY '0C8H';

    OUTPUT(PORTA) = CPU$INT$VECTOR(CPU);

    RETURN
END; /* EDWR$INT Procedure. */

/*************************************************************/
/*       Inner Traffic Controller Interface Modules        */
/*************************************************************/


/*************************************************************/
/*      IDLE$VP Procedure                                   */
/*---------------------------------------------------------*/
/*      Sets the state of the virtual processor now        */
/*      running to idle, binds the "Idle DBR", sets a      */
/*      high priority and calls Vp_Scheduler.              */
/*************************************************************/

IDLE$VP: PROCEDURE REENTRANT PUBLIC;

         DECLARE VP$TO$IDLE BYTE;

  VP$TO$IDLE = ITC$RET$VP;
  VPM(VP$TO$IDLE).VP$STATE = IDLE;
  VPM(VP$TO$IDLE).VP$PRIORITY = 10;
  VPM(VP$TO$IDLE).SS$REG = IDLE$DBR;
  CALL VPSCHEDULER;

  RETURN;

END; /* IDLE$VP Procedure. */

/*************************************************************/
/*      ITC$LOAD$VP Procedure                               */
/*---------------------------------------------------------*/
/*      Performs a "Swap Virtual DBR". Binds the virtual */
/*      processor to the new process, updates VP$PRIORITY */
/*      and SS$REG, and sets state to ready.               */
/*************************************************************/

ITC$LOAD$VP: PROCEDURE(PRI$PARM,DBR$PARM) REENTRANT PUBLIC;

             DECLARE PRI$PARM  BYTE,
                     DBR$PARM  WORD,
                     LOAD$VP   BYTE;
  /* Identify running virtual processor. */
  LOAD$VP = ITC$RET$VP;

  /* Bind the virtual processor. */
  VPM(LOAD$VP).VP$STATE = READY;
  VPM(LOAD$VP).VP$PRIORITY = PRI$PARM;
  VPM(LOAD$VP).SS$REG = DBR$PARM;

  /* Schedule the virtual processor. */
  CALL VPSCHEDULER;

  RETURN;

END; /* ITC$LOAD$VP Procedure. */

/**********************************************************/
/*      ITC$RET$VP Procedure                              */
/*--------------------------------------------------------*/
/*      Function call which returns the identity of the   */
/*      virtual processor which is now running on the     */
/*      physical processor.                               */
/**********************************************************/

ITC$RET$VP: PROCEDURE BYTE REENTRANT PUBLIC;

/* Search through the set of virtual processors assigned */
/* to the phsyical processor.                            */
RUNNING$VP$ID = VP$START;

DO WHILE /* Have not found the running virtual processor */
   (VPM(RUNNING$VP$ID).VP$STATE <> RUNNING);

   /* Search next entry. */
   RUNNING$VP$ID = RUNNING$VP$ID + 1;

END; /* While loop search for running VP.  RUNNING$VP$ID */
     /* points to the running virtual processor.         */


        /* Return the identity of the virtual processor  */
        /* in the AX (Accumulator) Register              */
RETURN RUNNING$VP$ID;

END; /* ITC$RET$VP Procedure. */

/**********************************************************/
/*      CHECKPREEMPT Procedure                            */
/*--------------------------------------------------------*/
```

```
/*     Checks for a virtual preempt pending.  If there     */
/*     is one, calls the Traffic Controller Preempt        */
/*     Handler.                                             */
/***********************************************************/

CHECKPREEMPT:  PROCEDURE REENTRANT PUBLIC;

                 DECLARE  RUNNING$VP BYTE;

/* Find the identity of the running virtual processor      */
RUNNING$VP = ITC$RET$VP;

DO WHILE /* Preempt Pending Flag of the virtual */
         /* processor is on.                    */
         VPM(RUNNING$VP).PE$PEND = TRUE;

         /* Reset Preempt Pending Flag. */
         VPM(RUNNING$VP).PE$PEND = FALSE;

         /* and call Traffic Controller Preempt Handler. */
         CALL TC$PE$HANDLER;

         RUNNING$VP = ITC$RET$VP;

END; /* While loop handling of the virtual preempt.        */

RETURN;

END; /* CHECKPREEMPT Procedure. */


/***********************************************************/
/*     ITC$SEND$PREEMPT Procedure                          */
/*---------------------------------------------------------*/
/*     Issues virtual preempts (preempt interrupts to      */
/*     virtual processors) by setting the appropriate      */
/*     Preempt Pending Flag in the Virtual Processor Map   */
/*     and then issuing a hardware interrupt if the        */
/*     processor is on a different physical processor.     */
/***********************************************************/

ITC$SEND$PREEMPT: PROCEDURE(TGT$CPU,VP$ID) REENTRANT PUBLIC;

                     DECLARE TGT$CPU WORD,
                             VP$ID   WORD;

/* SET THE PRE-EMPT PENDING FLAG */
VPM(TGT$CPU * VPS$PER$CPU + VP$ID).PE$PEND = TRUE;
```

```
IF (TGT$CPU <> CPU$NUMBER) THEN
  CALL HDWR$INT(TGT$CPU);

RETURN;
END; /* ITC$SEND$PREEMPT Procedure. */

/*****************************************************************/
/*      ITC$AWAIT Procedure                                      */
/*---------------------------------------------------------------*/
/*      Eventcount synchronization mechanism for use by the      */
/*      Inner Traffic Controller in the management of            */
/*      system resources.                                        */
/*****************************************************************/

ITC$AWAIT: PROCEDURE(EVC$ID,AWAITED$VALUE) REENTRANT PUBLIC;
           DECLARE EVC$ID          BYTE,
                   AWAITED$VALUE   WORD,
                   RUNNING$VP      BYTE,
                   I               BYTE;

  /* Lock the Virtual Processor Map */
  DO WHILE LOCKSET(@VPM$LOCK,119);
  END;

  DO;
    /* Identify the running virtual processor */
    RUNNING$VP = ITC$RET$VP;

    IF SYS$EVC$TABLE(EVC$ID) < AWAITED$VALUE THEN

      DO;
        VPM(RUNNING$VP).VP$STATE = WAITING;
        VPM(RUNNING$VP).EVC$AW$ID = EVC$ID;
        VPM(RUNNING$VP).EVC$AW$VALUE = AWAITED$VALUE;
     END;

    ELSE

      VPM(RUNNING$VP).VP$STATE = READY;
  END;

  /* Schedule the virtual processor. */
  CALL VPSCHEDULER;

  /* Unlock the Virtual Processor Map. */
  VPM$LOCK = 0;
  RETURN;
END; /* ITC$AWAIT Procedure. */

/*************************************************************/
/*     ITC$ADVANCE Procedure                                 */
/*-----------------------------------------------------------*/
/*     Eventcount signalling mechanism.  Used by the Inner */
/*     Traffic Controller in managing resources.            */
/*************************************************************/

ITC$ADVANCE: PROCEDURE(EVC$ID) REENTRANT PUBLIC
             DECLARE EVC$ID BYTE,
                     I      BYTE;

   /* Lock the Virtual Processor Map */
   DO WHILE LOCKSET(@VPM$LOCK,119);
   END;

   SYS$EVC$TABLE(EVC$ID) = SYS$EVC$TABLE(EVC$ID) + 1;

   DO I = 0 TO (NR$VPS - 1);
     IF VPM(I).EVC$AW$ID = EVC$ID THEN
       IF VPM(I).EVC$AW$VALUE <= SYS$EVC$TABLE(EVC$ID) THEN
         DO;
           VPM(I).VP$STATE = READY;
           VPM(I).EVC$AW$ID = 255;
           VPM(I).EVC$AW$VALUE = 0;
           IF (I < VP$START) OR (I > VP$END) THEN
             CALL HDWR$INT(I/VPS$PER$CPU);
         END;
   END;
   CALL VPSCHEDULER;

   /* Unlock the Virtual Processor Map */
   VPM$LOCK = 0;

   RETURN;
END; /* ITC$ADVANCE Procedure. */

/*************************************************************/
/*                  TRAFFIC CONTROLLER                       */
/*************************************************************/


/*************************************************************/
/*            External Global Data Declarations              */
/*************************************************************/
```

```
DECLARE APT(1) STRUCTURE
       (STATE            BYTE,
        AFFINITY         BYTE,
        VP$ID            BYTE,
        PRIORITY         BYTE,
        LOAD$THREAD      BYTE,
        EVC$VALUE$AW     WORD,
        THREAD           BYTE,
        DBR              WORD) EXTERNAL;

DECLARE APT$LOCK BYTE EXTERNAL;

DECLARE PROCESSES BYTE EXTERNAL;

DECLARE LOAD$LIST(16) BYTE EXTERNAL;

DECLARE EVC$TABLE(1) STRUCTURE
       (EVC$NAME(6)    BYTE,
        EVC$VALUE      WORD,
        APT$PTR        BYTE) EXTERNAL;

DECLARE EVENTS BYTE EXTERNAL;

DECLARE SEQ$TABLE(1) STRUCTURE
       (SEQ$NAME(6)    BYTE,
        SEQ$VALUE      WORD) EXTERNAL;

DECLARE SEQUENCERS BYTE EXTERNAL;

DECLARE (CPU$NUMBER,VP$START,VP$END,VPS$PER$CPU)
                        BYTE EXTERNAL;

DECLARE (NR$PHPS,NR$VPS) BYTE EXTERNAL;

DECLARE CPU$INT$VECTOR(16) BYTE EXTERNAL;

DECLARE PRO$PARAM     STRUCTURE
       (FLAGS        WORD,
        CS           WORD,
        IP           WORD,
        ES           WORD,
        DS           WORD,
        AX           WORD,
        CX           WORD,
        DX           WORD,
        BX           WORD,
        SI           WORD,
        DI           WORD,
          SS           WORD,
          PRIORITY     BYTE,
          AFFINITY     BYTE) EXTERNAL;

/*             Literal Constant Declarations                    */

DECLARE FALSE     LITERALLY   '0',
        READY     LITERALLY   '1',
        RUNNING   LITERALLY   '3',
        BLOCKED   LITERALLY   '7',
        TRUE      LITERALLY '119',
        NOT$FOUND LITERALLY '255',
        NIL       LITERALLY '255';

/*           External Procedure Declarations                    */

ITC$RET$VP: PROCEDURE BYTE EXTERNAL;
END;

ITC$LOAD$VP: PROCEDURE(PRI$PARM,DBR$PARM) EXTERNAL;
             DECLARE PRI$PARM BYTE,
                     DBR$PARM WORD;
END;

IDLE$VP: PROCEDURE EXTERNAL;
END;

ITC$SEND$PREEMPT: PROCEDURE(TGT$CPU,VP$ID) EXTERNAL;
                  DECLARE TGT$CPU WORD,
                          VP$ID   WORD;
END;


/***************************************************************/
/*                   THE PROCESS SCHEDULER                     */
/***************************************************************/

/**************************************************************/
/*      TC$SCHEDULER Procedure                                */
/*------------------------------------------------------------*/
/*      Process scheduler.  Searches for the highest          */
/*      priority runnable process to load onto the            */
/*      virtual processor.  If no runnable process is         */
/*      found, will idle the virtual processor.               */
/**************************************************************/

TC$SCHEDULER: PROCEDURE REENTRANT PUBLIC;
```

```
                    DECLARE PROCESS        BYTE,
                            SELECT$PROCESS BYTE;

PROCESS = LOAD$LIST(CPU$NUMBER);

SELECT$PROCESS = FALSE;

/* Search down Load$List for the highest priority        */
/* ready process runnable on this physical processor.    */
DO WHILE /* Have not found a runnable process.           */
   (SELECT$PROCESS = FALSE);

   IF /* Haven't reached the end of the Load$List */
     (PROCESS <> NIL) THEN

     DO; /* Check process. */

       IF /* Process is ready. */
         (APT(PROCESS).STATE = READY) THEN

          /* Select the process to run. */
          SELECT$PROCESS = TRUE;

       ELSE

          /* Check the next process. */
          PROCESS = APT(PROCESS).LOAD$THREAD;

    END; /* If then else. */

END; /* While loop search for next ready process. */

IF /* Have found a ready process to run. */
  (SELECT$PROCESS = TRUE) THEN

  DO; /* Give away the virtual processor. */
    APT(PROCESS).STATE = RUNNING;
    APT(PROCESS).VP$ID = ITC$RET$VP;
    CALL ITC$LOAD$VP(APT(PROCESS).PRIORITY,APT(PROCESS).DBR);
  END; /* Give away the virtual processor. */

ELSE

  /* No runnable process has been found so idle the      */
  /* virtual processor.                                  */
  CALL IDLE$VP;

  RETURN;
```

```
END; /* TC$SCHEDULER Procedure.*/

/**************************************************************/
/*            PROCESS SCHEDULER INTERNAL MODULES             */
/**************************************************************/

/**************************************************************/
/*     TC$LOCATE$EVC Procedure.                               */
/*------------------------------------------------------------*/
/*     Function call.  Returns the identity of the            */
/*     eventcount (the index of the eventcount in the         */
/*     Eventcount Table) in the AX (Accumulator)              */
/*     Register.  Input argument is a pointer to the          */
/*     byte array in the user process holding the name        */
/*     of the eventcount.                                     */
/**************************************************************/

TC$LOCATE$EVC: PROCEDURE(E$NAME$PTR) BYTE REENTRANT PUBLIC;

               DECLARE E$NAME$PTR POINTER;
               DECLARE CHAR BASED E$NAME$PTR (5) BYTE;
               DECLARE I      BYTE,
                       EVC$ID BYTE,
                       MATCH  BYTE;

I = Ø;
EVC$ID = Ø;
MATCH = FALSE;

/* Search down the eventcount table to locate the */
/* desired eventcount by matching the names       */

DO WHILE /* haven't found the eventcount and */
         /* haven't reached end of table     */
   (MATCH = FALSE) AND (EVC$ID < EVENTS);

   IF /* the two characters match */
     (CHAR(I) = EVC$TABLE(EVC$ID).EVC$NAME(I)) THEN

     DO; /* Check for end of strings */

       IF /* Reached the end of the strings */
         CHAR(I) = '%' THEN

         /* Have located the desired eventcount */
         MATCH = TRUE;
```

```
          ELSE

            /* Look at the next character */
            I = I + 1;

        END; /* Check for end of strings */

      ELSE

        DO; /* Ready for check next entry */

          I = 0;
          EVC$ID = EVC$ID + 1;

        END; /* Ready for check next entry */

   END; /* While loop search for desired eventcount */

   IF /* Have found the eventcount */
     (MATCH = TRUE) THEN

     /* Return its index in the EVC$TABLE */
     RETURN EVC$ID;

   ELSE

     /* Return NOT$FOUND error code */
     RETURN NOT$FOUND;

   END; /* TC$LOCATE$EVC Procedure. */

   /*************************************************************/
   /*      TC$LOCATE$SEQ Procedure                             */
   /*----------------------------------------------------------*/
   /*      Function call.  Returns the index in the            */
   /*      sequencer table of the sequencer name given         */
   /*      to it.  Input arguament is a pointer to the         */
   /*      string name of the sequencer in the application     */
   /*      program.                                            */
   /*************************************************************/

   TC$LOCATE$SEQ: PROCEDURE(S$NAME$PTR) BYTE REENTRANT PUBLIC;

                 DECLARE S$NAME$PTR POINTER;
                 DECLARE CHAR BASED S$NAME$PTR (5) BYTE;
                 DECLARE I      BYTE,
                         SEQ$ID BYTE,
                         MATCH  BYTE;
```

```
I = 0;
SEQ$ID = 0;
MATCH = FALSE;

/* Search down the sequencer table to locate the */
/* desired sequencer by matching the names.      */

DO WHILE /* Haven't found the sequencer and */
         /* haven't exhausted the list.     */
   (MATCH = FALSE) AND (SEQ$ID < SEQUENCERS);

   IF /* The two characters match. */
     (CHAR(I) = SEQ$TABLE(SEQ$ID).SEQ$NAME(I)) THEN

     DO; /* Check for end of strings. */

       IF /* Reached the end of the strings. */
         CHAR(I) = '%' THEN

         /* Have located the desired sequencer. */
         MATCH = TRUE;

       ELSE

         /* look at the next character. */
         I = I + 1;

     END; /* Check for end of strings. */

   ELSE

     DO; /* Ready for check of next entry. */

       I = 0;
       SEQ$ID = SEQ$ID + 1;

     END; /* Ready for check of next entry. */

END; /* While loop search for desired sequencer. */

IF /* Have found the sequencer. */
  (MATCH = TRUE) THEN

  /* Return its index in the SEQ$TABLE. */
  RETURN SEQ$ID;

ELSE
```

```
  /* Return NOT$FOUND error code. */
  RETURN NOT$FOUND;

END; /* TC$LOCATE$SEC Procedure. */

/*************************************************************/
/*           TRAFFIC CONTROLLER INTERFACE MODULES            */
/*************************************************************/

/*************************************************************/
/*      AWAIT Procedure                                      */
/*-----------------------------------------------------------*/
/*      Inter-process synchronization primitive.             */
/*      Suspends execution of the calling process until      */
/*      the event specified in the input argument            */
/*      'EVC$VAL$PARM' has occurred (the eventcount          */
/*      reaches this value).  The result is that the         */
/*      process will "give away" the virtual processor       */
/*      to which it is bound.                                */
/*************************************************************/

AWAIT: PROCEDURE(EVENTCOUNT,VALUE) REENTRANT PUBLIC;

       DECLARE    EVENTCOUNT    POINTER,
                  VALUE         WORD,
                  EVC$ID        BYTE,
                  CURRENT$VP    BYTE,
                  PROCESS       BYTE;

/* Assert global lock on the Active Process Table. */
DO WHILE LOCKSET(@APT$LOCK,119);
END;

/* Get identity of the virtual processor running on      */
/* physical processor.                                   */
CURRENT$VP = ITC$RET$VP;

/* Search the Active Process Table (by the Load List     */
/* to find the process bound to the running virtual      */
/* processor.                                            */
PROCESS = LOAD$LIST(CPU$NUMBER);

DO WHILE /* Haven't found the process bound to this vp. */
   (APT(PROCESS).VP$ID <> CURRENT$VP);

   /* Look at the next entry in the Load$List. */
   PROCESS = APT(PROCESS).LOAD$THREAD;
```

```
END; /* While loop search of Load$List. */

/* Get the EVC$TABLE index for this eventcount. */
EVC$ID = TC$LOCATE$EVC(EVENTCOUNT);

IF /* This process is to enter the blocked state. */
  EVC$TABLE(EVC$ID).EVC$VALUE < VALUE THEN

  DO; /* Set the required APT values. */

    APT(PROCESS).STATE = BLOCKED;
    APT(PROCESS).VP$ID = NIL;
    APT(PROCESS).EVC$VALUE$AW = VALUE;

    /* Add blocked process to head of blocked list. */
    APT(PROCESS).THREAD =  EVC$TABLE(EVC$ID).APT$PTR;

    /* Reset table pointer to the current process. */
    EVC$TABLE(EVC$ID).APT$PTR = CURRENT$VP;

  END; /* Do. Place process in the blocked state. */

ELSE

  /* If the event has already occurred, process will  */
  /* enter the ready state -- it will not be blocked. */
  APT(PROCESS).STATE = READY;
  APT(PROCESS).VP$ID = NIL;

  CALL TC$SCHEDULER;

  /* Unlock global Active Process Table Lock. */
  APT$LOCK = 0;

  RETURN;

END; /* AWAIT Procedure. */

/***********************************************************/
/*     ADVANCE Procedure                                   */
/*---------------------------------------------------------*/
/*     Used to signal the occurrence of a specified        */
/*     event. Increments the current value of the          */
/*     eventcount. Also signals all processes which        */
/*     are in the blocked state waiting for this event.    */
/***********************************************************/
```

```
ADVANCE: PROCEDURE(EVENTCOUNT) REENTRANT PUBLIC;

         DECLARE EVENTCOUNT    POINTER,
                 EVC$ID        BYTE,
                 PROCESS       BYTE,
                 PREV          BYTE,
                 PHP           BYTE,
                 VP            BYTE,
                 HI$PRI        BYTE,
                 PE$TO$SEND    BYTE,
                 PE$SENT       BYTE;

         DECLARE PE$PHP(16)  BYTE,
                 PE$VP(4)    BYTE;

/* Assert global lock on the Active Process Table. */
DO WHILE LOCKSET(@APT$LOCK,119);
END;

/* Increment the value of the eventcount by one. */
EVC$TABLE(EVC$ID).EVC$VALUE =
EVC$TABLE(EVC$ID).EVC$VALUE + 1;

/* Search Blocked List associated with the eventcount */
/* and unblock those processes waiting for this       */
/* event.  Set PROCESS to the first member of the     */
/* Blocked List.                                      */
PROCESS = EVC$TABLE(EVC$ID).APT$PTR;
PREV = PROCESS;

/* Initialize PE$PHP array. */
DO PHP = 0 TO NR$PHPS;
  PE$PHP(PHP) = FALSE;
END;

DO WHILE /* Not end of Blocked List. */
  PROCESS <> NIL;

  IF /* The event has already occured. */
    (EVC$TABLE(EVC$ID).EVC$VALUE >=
    APT(PROCESS).EVC$VALUE$AW) THEN

    DO; /* Unblock process (set state to ready), zero   */
        /* Eventcount Value Awaited entry of APT and     */
        /* flag the physical processor for preemption.   */
        APT(PROCESS).STATE = READY;
        APT(PROCESS).EVC$VALUE$AW = 0;
        PE$PHP(APT(PROCESS).AFFINITY) = TRUE;
```

```
          /* Remove process from the Blocked List. */
          IF /* First member of the Blocked List.  */
            (PREV = NIL) THEN

            /* Reset pointer around the deleted member. */
            EVC$TABLE(EVC$ID).APT$PTR =
            APT(PROCESS).THREAD;

          ELSE

            /* Set previous member's pointer around the */
            /* deleted process.                         */
            APT(PREV).THREAD = APT(PROCESS).THREAD;

      END; /* Do.  Remove process from Blocked List. */

      /* SEARCH NEXT ENTRY */
      PREV = PROCESS;
      PROCESS = APT(PROCESS).THREAD;

  END; /* While loop search of Blocked List. */

  DO /* Look for the PHP's with VP's to preempt. */
    PHP = 0 TO PHPS;

    IF /* PHP is flagged for a preempt. */
      PE$PHP(PHP) = TRUE THEN

      DO; /* Find VP's to preempt. */

      DO /* Flag all VP's for preemption. */
        VP = 0 TO VPS$PER$CPU;

        PE$VP(VP) = TRUE;

      END; /* Initialize PE$VP array. */

      HI$PRI= 0;
      PE$TO$SEND = 0;
      PROCESS = LOAD$LIST(PHP);

      DO WHILE /* Search down Load List to find those  */
               /* processes which should be running.   */
               /* Determine which VPs not to preempt.  */
        (PROCESS <> NIL) AND
        (HI$PRI < VPS$PER$CPU);
```

```
IF /* Found a process which should be running */
   /* that actually is running.        */
  APT(PROCESS).STATE = RUNNING THEN

  DO; /* Increment number found and do not  */
      /* preempt its VP.                    */

    HI$PRI = HI$PRI + 1;
    PE$VP(APT(PROCESS).VP$ID) = FALSE;

  END;

ELSE

  IF /* Found a process which should be running */
     /* but is in the ready state.              */
     APT(PROCESS).STATE = READY THEN

     DO; /* Increment number found and indicate  */
         /* that a preempt will have to be sent  */
         /* to get it running.                   */

       HI$PRI = HI$PRI + 1;
       PE$TO$SEND = PE$TO$SEND + 1;

     END;

END; /* While loop search of Load List. */

PE$SENT = 0; /* Used to keep track of the */
                        /* number of preempts sent.  */
VP = 0; /* Begin at first VP on the PEP. */

DO WHILE /* There are more preempts to send. */
   (PE$SENT <= PE$TO$SEND);

   IF /* A preempt is to be sent to this VP. */
     PE$VP(VP) = TRUE THEN

     DO; /* Issue the preempt and tally it. */

       CALL ITC$SEND$PREEMPT(PEP,VP);
       PE$SENT = PE$SENT + 1;

     END; /* Issue preempt. */

     /* Check the next VP. */
     VP = VP + 1;
```

```
        END; /* While loop send preempts. */

      END; /* While loop determine VPs to preempt. */

    END; /* Determine PHPs to preempt. */

    END;

    /* Ready the calling process. */

    /* Get identity of running VP. */
    VP = ITC$RET$VP;

    /* Search Load List Thread to find VP$ID match. */
    PROCESS = LOAD$LIST(CPU$NUMBER);

    DO WHILE /* Have not found process bound to this VP. */
       (APT(PROCESS).VP$ID <> VP);

       /* Look at next entry in Load List. */
       PROCESS = APT(PROCESS).LOAD$THREAD;

    END; /* While loop search of Load List. */

    /* Ready the calling process. */
    APT(PROCESS).STATE = READY;
    APT(PROCESS).VP$ID = NIL;

     CALL TC$SCHEDULER;

    /* Unlock Active Process Table. */
    APT$LOCK = 0;

    RETURN;
    END; /* ADVANCE Procedure. */

    /************************************************************/
    /*      TICKET Procedure                                    */
    /*----------------------------------------------------------*/
    /*      Function call.  Returns a unique sequencer value.   */
    /************************************************************/

    TICKET: PROCEDURE(SEQUENCER) BYTE REENTRANT PUBLIC;
            DECLARE SEQUENCER POINTER,
                    SEQ$ID    BYTE,
                    VALUE     WORD;
       /* Lock the Active Process Table. */
       DO WHILE LOCKSET(@APT$LOCK,119);
       END;

       /* Identify the sequencer. */
       SEQ$ID = LOCATE$SEQ(SEQUENCER);

       /* First obtain value to be returned to the caller */
       VALUE = SEQ$TABLE(SEQ$ID).SEQ$VALUE;

       /* Then increment the value of the sequencer */
       SEQ$TABLE(SEQ$ID).SEQ$VALUE =
       SEQ$TABLE(SEQ$ID).SEQ$VALUE + 1;

       /* Unlock the Active Process Table */
       APT$LOCK = 0;

       /* Return the value to the caller. */
       RETURN VALUE;
END; /* TICKET Procedure. */

/*************************************************************/
/*     READ Procedure                                        */
/*-----------------------------------------------------------*/
/*     Function call. Returns the current value of the       */
/*     eventcount specified in the call.                     */
/*************************************************************/

READ: PROCEDURE(EVENTCOUNT) BYTE REENTRANT PUBLIC;
      DECLARE EVENTCOUNT POINTER,
              EVC$ID     BYTE,
              VALUE      WORD;

    /* Lock the Active Process Table. */
    DO WHILE LOCKSET(@APT$LOCK,119);
    END;

    /* Identify the eventcount. */
    EVC$ID = LOCATE$EVC(EVENTCOUNT);

    /* "Read" the current value of the eventcount. */
    VALUE = EVC$TABLE(EVC$ID).EVC$VALUE;

    /* Unlock the Active Process Table. */
    APT$LOCK = 0;

    /* Return the current value to the caller. */
    RETURN VALUE;
```

```
END; /* READ Procedure. */

/**************************************************************/
/*      CREATE$EVC Procedure                                 */
/*-----------------------------------------------------------*/
/*      "Creates" an eventcount by making an entry for it    */
/*      in the eventcount table "EVC$TABLE" and setting      */
/*      the initial value of the eventcount to 0.            */
/**************************************************************/

CREATE$EVC: PROCEDURE(EVENTCOUNT) REENTRANT PUBLIC;
            DECLARE EVENTCOUNT POINTER;
            DECLARE CHAR BASED NAME (6) BYTE;
            DECLARE I BYTE;

/* Lock the Active Process Table */
DO WHILE LOCKSET(@APT$LOCK,119);
END;

IF /* The eventcount had not already been created */
  LOCATE$EVC(EVENTCOUNT) = NOT$FOUND THEN

  DO;
    I = 0;
    DO /* Copy the name into EVC$TABLE */
       WHILE (CHAR(I) <> '%') AND (I < 5);

       /* Copy the character into the table. */
       EVC$TABLE(EVENTS).EVC$NAME(I) = CHAR(I);

     END; /* While loop. */

     /* Insert the delimiter '%' in the table entry. */
     EVC$TABLE(EVENTS).EVC$NAME(I) = '%';

     /* Increment EVENTS to indicate a new addition. */
     EVENTS = EVENTS + 1;

  END; /* Create the eventcount. */

/* Unlock the Active Process Table. */
APT$LOCK = 0;

RETURN;
END; /* CREATE$EVC Procedure. */

/**************************************************************/
/*      CREATE$SEQ Procedure                                 */
```

```
/*------------------------------------------------------------*/
/*     "Creates" a sequencer by establishing an entry in      */
/*     the sequencer table "SEQ$TABLE" and sets the            */
/*     initial value to Ø.                                    */
/**************************************************************/

CREATE$SEC: PROCEDURE(SEQUENCER) REENTRANT PUBLIC;
            DECLARE SEQUENCER POINTER;
            DECLARE CHAR BASED NAME (6) BYTE;
            DECLARE I BYTE;

/* Lock the Active Process Table */
DO WHILE LOCKSET(@APT$LOCK,119);
END;

IF /* The sequencer had not already been created */
  LOCATE$SEQ(SEQUENCER) = NOT$FOUND THEN

  DO;
    I = Ø;
    DO /* Copy the name into SEQ$TABLE */
      WHILE (CHAR(I) <> '%') AND (I < 5);

      /* Copy the character into the table. */
      SEQ$TABLE(SEQUENCERS).SEQ$NAME(I) = CHAR(I);

    END; /* While loop. */

    /* Insert the delimiter '%' in the table entry. */
    SEQ$TABLE(SEQUENCERS).SEQ$NAME(I) = '%';

    /* Increment SEQUENCERS to indicate a new addition. */
    SEQUENCERS = SEQUENCERS + 1;

END; /* Create the sequencer. */

/* Unlock the Active Process Table. */
APT$LOCK = Ø;

RETURN;
END; /* CREATE$SEC Procedure. */

/**************************************************************/
/*     CREATE$PROCESS Procedure                               */
/*------------------------------------------------------------*/
/*     "Creates" a process by initializing its stack and      */
/*     initializing an entry for it in the Active Process     */
/*     Table.                                                 */
```

```
/*****************************************************************/

CREATE$PROCESS: PROCEDURE(PPB$PTR) REENTRANT PUBLIC;
                DECLARE PPB$PTR POINTER;
                DECLARE INIT$STACK$FRAME STRUCTURE
                       (FL   WORD,
                        CS   WORD,
                        IP   WORD,
                        ES   WORD,
                        DS   WORD,
                        AX   WORD,
                        CX   WORD,
                        DX   WORD,
                        BX   WORD,
                        SI   WORD,
                        DI   WORD,
                        RET  WORD,
                        BP   WORD,
                        SP   WORD);

                DECLARE INTERRUPT LITERALLY '119';

/* Lock the Active Process Table. */
DO WHILE LOCKSET(@APT$LOCK,119);
END;

  /* Set up initialization stack frame. */
  INIT$STACK$FRAME.FL = PRO$PARAM.FL;
  INIT$STACK$FRAME.CS = PRO$PARAM.CS;
  INIT$STACK$FRAME.IP = PRO$PARAM.IP;
  INIT$STACK$FRAME.ES = PRO$PARAM.ES;
  INIT$STACK$FRAME.DS = PRO$PARAM.DS;
  INIT$STACK$FRAME.AX = PRO$PARAM.AX;
  INIT$STACK$FRAME.CX = PRO$PARAM.CX;
  INIT$STACK$FRAME.DX = PRO$PARAM.DX;
  INIT$STACK$FRAME.BX = PRO$PARAM.BX;
  INIT$STACK$FRAME.SI = PRO$PARAM.SI;
  INIT$STACK$FRAME.DI = PRO$PARAM.DI;
  INIT$STACK$FRAME.RET = INTERRUPT;
  INIT$STACK$FRAME.BP = 0;
  INIT$STACK$FRAME.SP = 6;

  /* Move initialization stack frame into memory. */
  MOVB(@INIT$STACK$FRAME,PPB.DBR,28);

  /* Enter process in Active Process Table. */
  APT(PROCESSES).STATE = PPB.STATE;
  APT(PROCESSES).AFFINITY = PPB.AFFINITY;
  APT(PROCESSES).VP$ID = NIL;
  APT(PROCESSES).PRIORITY = PPB.PRIORITY;
  APT(PROCESSES).EVC$VALUE$AW = 0;
  APT(PROCESSES).THREAD = NIL;
  APT(PROCESSES).DBR = PPB.DBR;

  /* Enter process in the Loaded List by priority */
  PREV = NIL;
  NEXT = LOAD$LIST(CPU$NUMBER);
  DO WHILE PPB.PRIORITY > APT(NEXT).PRIORITY;
    PREV = NEXT;
    NEXT = APT(NEXT).LOAD$THREAD;
  END;
     IF NEXT = NIL THEN
       APT(PREV).LOAD$THREAD = ENTRY;
     ELSE
       IF NEXT = LOAD$LIST(CPU$NUMBER) THEN
         DO;
           APT(ENTRY).LOAD$THREAD =
              LOAD$LIST(CPU$NUMBER);
           LOAD$LIST(CPU$NUMBER) = ENTRY;
       ELSE
         DO;
           APT(PREV).LOAD$THREAD = ENTRY;
           APT(ENTRY).LOAD$THREAD = NEXT;
         END;

/* Unlock the Active Process Table. */
APT$LOCK = 0;

RETURN;
END; /* CREATE$PROCESS Procedure. */

/************************************************************/
/*      TC$PE$HANDLER Procedure                             */
/*----------------------------------------------------------*/
/*      Handles preempt interrupts.  Called by the          */
/*      Traffic Controller in response to a virtual         */
/*      preempt interrupt.  This module serves as the       */
/*      virtual interrupt entry point into the Traffic      */
/*      Controller.                                         */
/*         =========> Constitutes a loop. <=========        */
/************************************************************/

TC$PE$HANDLER: PROCEDURE REENTRANT PUBLIC;

/* Lock the Active Process Table. */
DO WHILE LOCKSET(@APT$LOCK,119);
```

```
END;

  CALL TC$SCHEDULER;

/* Unlock the Active Process Table. */
APT$LOCK = 0;
RETURN;

END; /* TC$PE$HANDLER Procedure. */
```

BIBLIOGRAPHY

1. Anderson, G. A. and Jensen, E. D., "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples," Computing Surveys, v. 7, no. 4, p. 197-213, December 1975.

2. Daley, R. C. and Dennis, J. B., "Virtual Memory, Processes, and Sharing in Multics," Communications of the ACM, v. 11, p. 306-312, May 1968.

3. Dijkstra, E. W., "Cooperating Sequential Processes," in Programming Languages, F. Guneys, ed., Academic Press, 1968.

4. Horning, J. J. and Randell, B., "Process Structuring," Computing Surveys, v. 5, no. 1, p. 5-30, March 1973.

5. Intel Corporation, PL/M-86 Programming Manual, 1978.

6. Intel Corporation, ISBC 86/12A Single Board Computer Hardware Reference Manual, 1979.

7. Intel Corporation, MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users, 1979.

8. Intel Corporation, MCS-86 Macro Assembly Language Reference Manual, 1979.

9. Intel Corporation, MCS-86 Macro Assembler Operating Instructions for ISIS-II Users, 1979.

10. Intel Corporation, ISIS-II PL/M-86 Compiler Operator's Manual, 1979.

11. O'Connell, J. S. and Richardson, L. D., Distributed, Secure Design for a Multi-Microprocessor Operating System, Naval Postgraduate School, 1979.

12. Organick, E. I., The Multics System: An Examination of Its Structure, The MIT Press, Cambridge, Massachusetts, 1972.

13. Reed, D. P. and Kanodia, R. J., "Synchronization with Eventcounts and Sequencers," Communiations of the ACM, v. 22, p. 115-123, February 1979.

14. Reed, D. P., Processor Multiplexing in a Layered Operating System, M.S. Thesis, Massachusetts Institute of Technology, MIT/LCS/TR-164, 1976.

15. Reitz, S. L., An Implementation of Multipogramming and Process Management for a Security Kernel Operating System, M.S. Thesis, Naval Postgraduate School, 1980.

16. Ross, J. L., Design of a System Initializaton Mechanism for a Multiple Microcomputer, M.S. Thesis, Naval Postgraduate School, 1980.

17. Saltzer, J. E., Traffic Control in a Multiplexed Computer System, Ph.D. Thesis, Massachusetts Institute of Technology, 1966.

18. Swann, R., Fuller, S., and Siewiorek, D., "Cm*: a Modular, Multi-microprocessor," in Proceedings National Computer Conference, p. 637-644, AFIPS, 1977.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Asst. Professor U. R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
5.	LT.COL. R. R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	Professor T. F. Tao, Code 62Tv Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	2
7.	LT Warren J. Wasson, USN Commander Naval Electronics Systems Command PME 124 Washington, D.C. 20360	3
8.	CAPT J. L. Ross 552 AWACW/ADM Tinker AFB, Oklahoma 73145	1
9.	LCDR S. L. Reitz NAVSEA TECH REP St. Paul, Minnesota 30845	1
10	LT P. A. Myers NAVDAC Washington Navy Yard Washington, D.C. 20374	1

DA

FIL