INTEGRATING ARCHITECTURE AND OPERATING SYSTEMS*

by

G. J. Nutt
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

#CU-CS-104-77

March, 1977
revised September, 1977

* This work supported under NSF grant No. MCS74 08328 A01.

ABSTRACT

Recent advances in integrated circuit technology have drawn the fields of architecture and operating systems designs closer together. In particular, bit slice bipolar microprocessor technology provides a medium for realistic integration of many operating system functions directly into firmware and hardware. Some aspects of this integration are illustrated through two examples: memory relocation and interprocess communication mechanisms.

INTRODUCTION

Current trends in integrated circuit technology are causing multiple processor computer systems to become increasingly more attractive to the computer system designer, [6,8,10]. The basic design atoms have advanced from vacuum tubes, through transistors to large scale integrated packages which incorporate several hundred gates per chip, [7]. There are at least two important points resulting from these advances that intimately affects operating system design. First, the ability to economically produce parallel hardware is rapidly outpacing the ability to produce software to control such hardware. Questions of controlling parallelism in a system have been answered (at least to the point of finding theoretical solutions and mechanisms to control parallel processes). However, design technologies for operating systems do not seem to be so well understood that one can implement an operating system for a multiple processor machine in a manner that corresponds to the theoretical solutions. The second observation has to do with the integration of hardware and software designs. Since the basic design atoms have become more powerful, the area of logical design has become smaller; circuit designers incorporate larger portions of logical designs within each chip, and software designers can now program these chips to achieve "hardware" effects. (Logic designers are still necessary to interconnect the large scale integrated circuits). Designing computer systems based on current LSI technology allows many of the architectural features to be at least partially controlled by the software designer. This participation in hardware design is deeper than microprogramming a fixed architecture machine, since the operating system designer should now be able to understand the atoms and be able to work with a logic designer to build the kind of hardware system most viable for his software

designs.* As the packaging density for components increases, and very large scale integration reaches production, the trend will continue to an extreme integration of software and hardware.

One particular LSI development that will have an impact on the design of large word size machines is that of the bit slice bipolar microprocessor, e.g. Intel 3000 series, AMD 2900 series, etc. A central processing unit constructed from these chips is not as compactly packaged as is the case with the more widely known Intel 8080 or Motorola MC6800 microprocessors, but such a CPU is faster and more flexible. The basic chips required to construct a CPU include a sequencer (or control unit), an arithmetic-logical unit, and a control store. Each bit slice ALU can perform a few arithmetic and logical operations on 2-4 bit operands. The ALU will contain logic to perform the operations, to shift operands, and to incorporate a bank of internal registers. Multiple ALU can be interconnected to form a single ALU that operates on words with 64 or more bits. These resulting ALUs perform relatively fast arithmetic operations, since carry lookahead logic can be used to combine the individual bit slices.

The control unit of a bit slice microprocessor may have a fixed design, (as in the case of the Intel 3001), or it may also be expandable in a manner similar to the ALU , (as in the case of the AMD 2909). The first option restricts the flexibility of design, and the second option forces the designer to provide more chip interconnections. In either case, the control unit is microprogrammed, and it can be used to economically implement any of a wide variety of instruction sets once the data paths for the hardware have been established.

Bit slice bipolar microprocessors have typical microinstruction cycle times in the range of 100-200 ns. A register-register integer

*This integration of software and hardware at design time is in contrast to the approach taken in, e.g., the Venus operating system study [4], where the microprogrammed architecture was still primitive to software development.

add operation may require as little as one microinstruction for word widths of arbitrary size. The cost of bit slice microprocessor chips varies from about $5/bit (for the control unit and ALU) up to about $15/bit excluding the cost of control store.

The flexibility, speed, and cost all combine to make bit slice microprocessors a viable implementation tool for multiple processor computer systems with large word sizes. The simplicity of design allows the operating system to be easily implemented across software, firmware, and hardware boundaries. The remainder of this paper intends to illustrate the flexibility of operating systems design by discussing the organizations of a simple memory relocation mechanism and of an interprocess communication scheme. The first example is primarily one illustrating the applicability of microprogramming. This example could apply to any commercially-available user microprogrammable machine. The second example is specific to bit slice microprocessor implementation of an extended unit of a machine. There have been other applications in which a fixed instruction set microprocessor has been used to implement a single, well-defined function of a system, (e.g., see [6]). In this example, the extended unit is used to implement an entire set of functions on a data structure. This approach is used to implement a hardware/firmware instance of a monitor, [3]; the data structure is local to a set of functions that are implemented in the extended unit. The data in the local structure is not manipulated by any mechanism other than the extended unit itself; the extended unit hides the details of the local data structures from all other parts of the machine.

For this discussion, it is assumed that a multiple processor computer system composed of n processors, each of which supports m levels of multiprogramming is to be implemented as a set of bit slice

microprocessors; (this architecture is a generalization of a SIMD design, [5]). The processors share a memory system as shown in Figure 1, i.e. the memory is divided into 2n physically independent modules. Each processor has a private path to/from one module in the memory system. It also has a path over a shared memory bus to all other modules in the memory system. Each processor, P_i, is intended to make the majority of its memory references to memory module M_i; in the case that memory contents are shared among two or more processors, then no more than one of them can have a direct connection to the corresponding memory module. Memory modules M_n through M_{2n-2} are explicitly for shared use, and module M_{2n-1} is primarily used by a special purpose execution unit called the Processor Interface, PI.

In a multiple processor system, the operating system could be executed on a dedicated processor as in the case of the PRIME system design, [2]; or it might be distributed across the set of all processors in the system as is the case with the HYDRA operating system for the C.mmp, [9]. The distributed approach is favored here for the following reasons: Processors should be treated as identical system resources by the operating system so that if any single processor fails, the system will continue to operate at a possibly reduced rate. Distribution tends to make more effective use of the processor resource than the case where one processor is dedicated to the operating system. However, dedicated processor operating systems may be easier to implement and may also be more effective in terms of protection and process synchronization. For the distributed approach, operating system code and tables are stored in physical memory modules M_n through M_{2n-2}.

MEMORY RELOCATION

Hardware relocation registers can be used to advantage in systems that employ single segment, dynamically relocatable memory organizations as well as the more general virtual memory strategies. As long as the run time image of a program contains addresses relative to the first location loaded, then a base register can be used to relocate all addresses at the time of reference. Multiple segment strategies require multiple base registers (or a segment table containing the corresponding segment base register contents).

Given the physical memory arrangement shown in Figure 1, the program(s) executed on processor P_i should normally be loaded in memory module M_i. In some cases M_i will not be large enough to hold the program(s) and data for P_i, thus memory from some other module must be allocated to the corresponding process(es). There are two obvious strategies that might be used for memory allocation: Provided that the memory spaced required by the n processors does not exceed the amount of memory in M_0-M_{n-1}, and the number of processors requiring more memory than exists in their preferred modules is small, then space from M_{i-1}, M_i, and M_{i+1} can be allocated to P_i. This preserves the single segment address space, and requires only one base register per process for memory relocation. It is easy to construct examples that will introduce excessive memory conflict with this strategy. An alternative is to divide the memory space per process into two segments (logically one segment) such that the first portion of the logical segment is loaded in the preferred module, and the second portion is loaded into contiguous locations in modules M_n through M_{2n-2}. This strategy requires, as a minimum, two additional registers in each processor: A first-physical-segment base register (as before), a first-

physical-segment-length register, and a second-physical-segment base register to point to the second portion of the address space in the shared portion of the memory. If relocation is to be implemented in hardware, different mechanisms are required for the two strategies; if relocation is performed in software, then two different loader algorithms must be used, and programs cannot be easily moved within the memory. Alternatively, the relocation mechanism can be implemented in microprograms within each processor.

Typical bit slice microprocessors include from 10 to 16 registers within the ALU chip; some of these registers are used as general purpose registers available to the assembly language programmer, while the other may be used by the microprogrammer to implement various machine registers such as the program counter, instruction word register, etc. Microprograms may also require scratch registers for implementing certain machine level instructions. In some cases, the number of registers within the microprocessor ALU may be insufficient to satisfy all of these needs; in these cases, high speed 16 word random access memories can be used by the ALU even though the corresponding registers are physically external to the ALU chips, or writeable control store memory space can be used. This freedom of register usage at the microprogram level allows one to reconfigure the "abstract machine" used by the assembly language programmer. In particular, microprograms can be written to incorporate a single base register for address calculations; a base and a length register; two base registers and a length register, etc. by altering the microprograms.

The memory relocation algorithms, themselves, are easily determined by the microprograms. Effective address formation is implemented in microcode so that indexing and/or indirection correspond to micro-

subroutines used whenever appropriate. The memory relocation schemes mentioned above do not differ substantially from index register calculations and could possibly even use the same microsubroutines (with appropriate bound checking). The additional time required for firmware relocation is approximately 2 microcycles in the single segment memory case.

It may occasionally be necessary for a processor to generate an absolute memory address, e.g. to reference locations in memory module M_{2n-1}. In the hardware case, it may be difficult (i.e. expensive) to allow processors to sometimes reference memory via a relocation unit and at other times to bypass the unit. In the firmware case the desired flexibility is easily incorporated into the address formation microprograms by a simple conditional test. It is also possible to allow this ability to generate absolute addresses only for certain instructions such as privileged instructions.

Although somewhat slower than hardware relocation mechanisms, the generality and flexibility of the approach allows the operating system designer to control a wide variety of memory relocation schemes that are relatively secure from user programs.

PROCESS COMMUNICATION

In order for two or more processes to share resources or otherwise cooperatively execute in any manner, a mechanism for communication among processes must be established. There are a number of schemes for accomplishing processor communication in multiple processor systems. The simplest schemes use mailboxes and polling (e.g. in the Control Data 6000 series operating systems). If elapsed time is import-

ant during such communications, this method may be ineffective, since the average response time to inspect mailbox will increase with the number of processes, m×n, in the system. A more prompt scheme may use interrupts and mailboxes, but this scheme may require $O(n^2)$ lines interconnecting the n processors and the ability of a process to map a receiving process's name into the processor identification before issuing an interrupt. The number of lines can be sacrificed in exchange for response time by using a shared interrupt bus, [8]. A third scheme is to dispense with mailboxes, and to use multiple interrupts to distinguish between message types. This is essentially the approach taken in HYDRA, [9].

The communication scheme given here is an example of bit slice microprocessor technology applied to operating system and hardware integration. The scheme uses n interrupt lines for the n processor system, and message response time can be adjusted by priority assignments among processes and/or processors. The Processor Interface is used to implement a firmware form of communication among processes that is reminiscent of the software scheme used in the RC4000 Multiprogramming System, [1]. Notice, that the PI is a firmware/hardware implementation of a software notion; it is an integration of all technologies to achieve a goal.

One software method for interprocess communication in a multiprogrammed system is to include a kernel routine that is invoked by a sending process whenever it wishes to send a message to some other process. The kernel routine must find a process descriptor for the receiver process, possibly check to see if the sender has permission to transmit a message to the given receiver, and then attach the message to a queue within the descriptor of the designated receiver process. The sending process is then resumed by the kernel routine. It is usually not possible to implement hardware message passing of this

form in a multiprogrammed system, since the receiver process may not be executing on a processor at the time a message is directed toward it; the process identifier exists only in a descriptor and not in a hardware register. This software approach forces all authorization checks, message queuing, etc. to be susceptible to software bugs and volatility (assuming the kernel code is stored in read/write memory). A firmware approach to the problem allows the flexibility of the software method and the speed and safety of a hardware method. The following method for implementing communication mechanisms is an example of the approach.

Let each of the n processors contain a register made up of h bits, called the ID register, used to identify a process; (each process, in turn, has an ID register content). Let $ID_{ij}[k]$ denote bit k ($0 \leq k < h$) of process i ($0 \leq i < m$) on processor j ($0 \leq j < n$). If

$$ID_{ij}[k] \wedge ID_{i'j'}[k]=1 \quad \text{for some } 0 \leq k < h$$

then process i can <u>cooperatively communicate</u> with process i', and vice versa. Otherwise no communication is possible between process i and i'. This definition implies that a check on process identifiers is made in order to test for authorized access (in an unspecified mode). If process i can cooperatively communicate with process i' and additionally,

$$ID_{i'j'}[k]=1 \Longrightarrow ID_{ij}[k]=1 \quad \text{for all } 0 \leq k < h$$

then process i has <u>privilege</u> with respect to process i', (no two processes may have identical ID register contents). The concept of privilege is not symmetric as is cooperative communication. Privilege is used to establish a hierarchy (partial ordering) on the set of processes that exist at any given time within the system.

For example, suppose h=4 and there exist four processes in a system with two processor such that

ID_{00}=1111

ID_{01}=1100

ID_{10}=0011

ID_{02}=0001

Denote process i on processor j as (i,j), where the process index i is also the index of a multiprogrammed level within processor j. Then, (0,0) can cooperatively communicate with any other process. (0,1) can cooperatively communicate only with (0,0); this does not preclude indirect communication between (0,1) and, say, (1,0) through (0,0). (1,0), (0,2), and (0,0) can cooperatively communicate among themselves. (0,0) has privilege over all other processes in this example, and (1,0) has privilege with respect to (0,2).

The important component in determining the communicative ability of a process is the content of its ID register. Therefore, the values loaded into $ID_{i'j'}$ by process i on processor j is restricted such that

$$ID_{i'j'}[k] \leftarrow ID_{ij}[k]-y$$

where

$$0 \leq y \leq ID_{ij}[k].$$

i.e., process i cannot specify a new ID content for process i' where more bits are set in $ID_{i'j'}$ than are set in ID_{ij}. Furthermore, process i can change the content of $ID_{i'j'}$ only if process i has privilege with respect to process i'. Notice that this does not preclude the case where i=i' and j=j'. In any case, the process is prevented from increasing its communicative power without the help of some other process

that "supervises" the given process.

Each processor also contains an h bit signal register, S, used to indicate the ID of some process to which a communication is directed. That is, S is loaded with the ID of the receiver process before the sending process executes a send message instruction.

Two forms of message passing are used in the system: Active and passive communication. Active communication, called preemption, forces the receiving process to terminate execution of its current program and to begin execution of a program at the memory location specified within a preempt instruction executed in behalf of the sending process. Preemption of process i' by process i is allowed only if process i has privilege over process i'. Passive communication, called signaling, causes the receiving process to interrupt execution of the program it is currently executing and to begin executing a new program loaded at the absolute address pointed to by an interrupt register, INT, associated with the receiving process. Process i can signal process i' only if the two processes can cooperatively communicate.

Both forms of message passing act as interrupts for the receiving process, the differences between the two being the conditions under which the interrupt can be generated by the sender, and the control of the location at which the interrupt forces execution. During passive communication, the sender does not necessarily dictate the location which the receiver must branch to upon accepting the interrupt.

As with all forms of interrupts, the critical section problem arises in which a process cannot be interrupted during the execution of some code segment. Two additional 1 bit registers are included with each process(or) in order to turn off interrupts of either type during critical sections. The PA_{ij} register for process i executing on pro-

cessor P_j is set (true) if a privilege interrupt can be accepted by process i. CA_{ij} for process i on P_j is set (true) if a cooperative interrupt, i.e. a signal operation, can be accepted by process i. PA_{ij} and CA_{ij} are binary semaphores to protect critical sections within process i.

The Processor Interface, PI, is a bit slice microprocessor used to implement the process communication instructions of the repertoire of each processor; it is not used as a dedicated processor on which to implement an operating system. PI is invoked by a processor, P_j, whenever P_j stores a process communication instruction into a prespecified location in module M_{2n-1}. The ability of process i executing on processor j to reference this location is checked by the microcode within processor j. Each P_j has its own unique PI instruction buffer, and PI will service simultaneous requests as specified by any strategy provided by PI's microprogram, (e.g. round robin, FCFS, etc.). The processor that requests PI service is blocked (waiting for an interrupt) until the PI has finished executing the desired instruction. Upon instruction completion, PI sends an interrupt back to the requesting P_j.

In order to correctly implement signal and preempt instructions (as discussed above), PI must be able to reference the ID, S, INT, PA, and CA registers for each process on each processor. PI must be able to reference these registers in the abstract machine that executes a process whether the process is physically running or not. To integrate this (usually software implemented) function into the firmware, these register contents are stored in memory module M_{2n-1} rather than in an operating system-dependent process descriptor. Since signal and preempt instructions will potentially cause more than one message to be directed toward a given process before it is willing to accept any

message, (i.e., PA and CA are reset), message queues also need to be handled by the PI. One possible memory organization (data structure) to be used to describe this portion of a process's state is shown in Table 1. Locations I through I+n-1 are used as the mailbox for processor P_j to pass a communication instruction to PI. Locations J through J+n-1 are used to save the program counter content of P_j when it is interrupted by a preemption, (more details of the operation are given below). K through K+n-1 are used to save the program counter of P_j when it is interrupted by a signal. Memory locations L through L+4mn-4 contain a partial process descriptor for each process in the system. The form of the process descriptor for process i executing on processor j is shown in Figure 2. An unspecified number of memory locations (from Q to L) are used to hold messages of the form shown in Figure 3 in the queue. In order to see how this data structure is used to execute communication instructions, next consider some PI microprogram algorithms.

In the following, assume that there exist microprogram subroutines ENQUEUE and DEQUEUE to handle insertions and deletions to and from the message queues. The preemption instruction can be implemented as follows, (assume that process i on processor j executes the instruction):

```
if (ID_i'j'=S_ij) ∧ (ID_i'j' ⊆ ID_ij) then
comment process i has privilege with respect to process i';
   begin
      if "i' not running on j'" ∨ ¬ PA_i'j' then ENQUEUE
         else
            begin
               PA_i'j':=false;
               comment P_j' completes current instruction cycle;
               comment PC_j' is the program counter in P_j'
               MEMORY [J+j']:=PC_j';
               PC_j':= <effective address of PREEMPT instruction>;
               A-REGISTER_i'j' := A-REGISTER_ij;
               PA_ij := true
            end
   end;
comment Return instruction-complete interrupt to P_j;
```

Now, the operating system scheduler must issue a command to the PI whenever it schedules a process onto a processor. The effect of that instruction is not only to restore ID, S, and INT registers, but also to DEQUEUE pending messages for the schedule process.

The signal instruction is similar to preempt, except that the interrupt address is taken from the interrupted process's INT register, and the conditions under which one process can signal another (cooperative communication) are different from those for preemption (privileged communication). One possible algorithm for the signal instruction executed by process i is the following:

```
if (ID_i'j'=S_ij) ∧ (ID_i'j' ∩ ID_ij>0) then
comment processes i and i' can communicate;
   begin
      if "i' not running on j'" ∨¬ CA_i'j' ∨¬ PA_i'j' then ENQUEUE
         else
            begin
               PA_i'j' := CA_i'j' := false;
               comment P_j' completes current instruction cycle;
               MEMORY [K+j'] := PC_i'j';
               PC_i'j' := INT_i'j';
               A-REGISTER_i'j' := A-REGISTER_ij;
               PA_ij := true
            end
   end;
comment  Return instruction-complete interrupt to P_j;
```

Again, a scheduling or "restore state" command must be executed by the PI whenever processor is multiplexed in order to remove queue entries.

The PI executable instructions must also include those that load/store the various registers; most of these instructions are privileged instructions, and the load instructions have to check for legal contents whenever the ID register receives new data.

The mechanism suggested here can be viewed as a hardware implementation that strongly resembles Hoare's monitors, [3]. The data structures used by the Processor Interface are not available to any processes other than those that execute on the PI. Therefore, there particular configuration is of importance only to the PI microprograms and not to the remaining processes and processors in the machine. The

microprograms themselves are analogous to monitor procedures. Notice also that since PI is a single physical processing unit with the unique ability to execute process communication instructions, then the problems of mutual exclusion of processes from executing certain instructions (i.e. kernel routines) is taken care of by guaranteed sequential execution of PI.

SUMMARY

Large scale integrated circuit technology has advanced to the state in which the basic design atoms for building computers have begun to resemble the processor itself. Bit slice microprocessor sets offer the ability to build large word size machines interconnected in very general ways, where each machine is microprogrammed. The level of these logical building blocks allows the operating system designer to actively participate in the architecture of a machine in which he will develop software. The result is that the time has come when operating system functions can be implemented in hardware or firmware as a common design practice; it is no longer necessary to hypothesize that a given feature could be implemented in firmware or hardware and then implement it in software because of cost considerations.

This aspect of design has been illustrated by providing two examples of the incorporation of operating system tasks into the firmware and hardware. The memory relocation example relies on the microprogrammability of the processors in order to have flexibility in choosing a virtual memory strategy. The process communication example is a much more general application of both microprocessors as heterogeneous units of a multiple processor and as flexible units to implement the concept of a monitor.

ACKNOWLEDGEMENT

The author wishes to thank the National Science Foundation for the support of this work under grant number MCS74-08328 A01.

REFERENCES

[1] Brinch Hansen, P. (editor), RC4000 Software Multiprogramming System, A/S Regnecentralen, Copenhagen, Denmark, (Feb., 1971). 159 pages.

[2] Fabry, R. S., "Dynamic Verification of Operating System Decisions", Communications of the ACM, Vol.16, No.11, (Nov., 1973), pp. 659-668.

[3] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", Communications of the ACM, Vol.17, No.10, (Oct., 1974), pp. 549-557.

[4] Liskov, B. H., "The Design of the Venus Operating System", Communications of the ACM, Vol. 15, No. 3, (March, 1972), pp.140-143.

[5] Nutt, G. J., "Microprocessor Implementation of a Parallel Processor", Proceedings of the 4th Annual Symposium on Computer Architecture, (March, 1976), pp. 147-152.

[6] Radoy, C. H. and Lipovski, G. J., "Switched Multiple Instruction, Multiple Data Stream Processing", Proceedings of the 2nd Annual Symposium on Computer Architecture, (Jan., 1975), pp. 183-187.

[7] Verhofstadt, P.W.J., "Technology for Microprocessor Hardware", IEEE COMPCON 76, (Feb., 1976), pp. 19-22.

[8] Widdoes, L. D., Jr., "The Minerva Multi-Microprocessor", Proceedings of the 3rd Annual Symposium on Computer Architecture, (Jan., 1976), pp. 34-39.

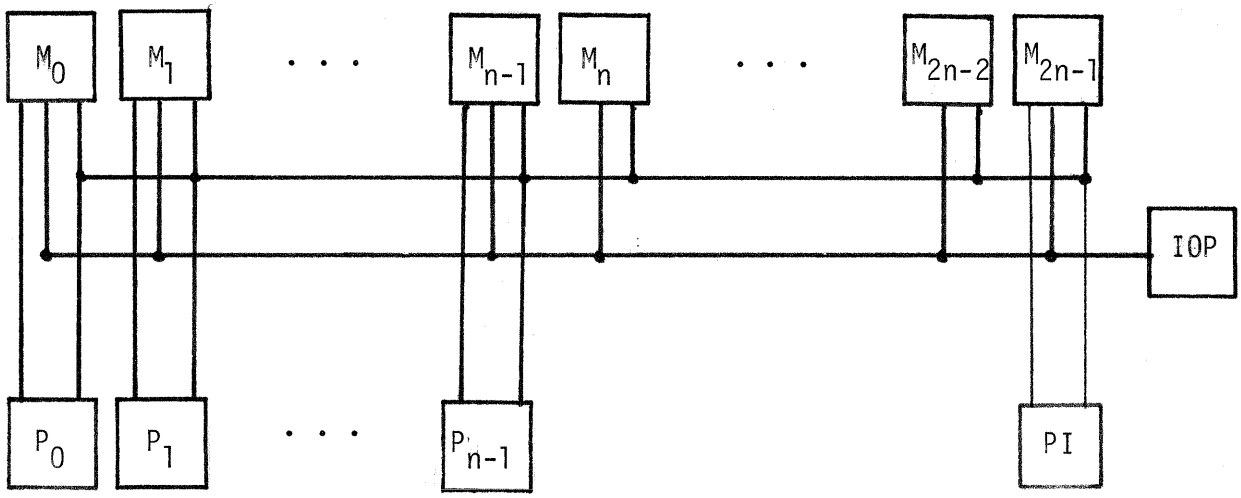
[9] Wulf, W. et al, "HYDRA: The Kernel of a Multiprocessor Operating System", Communications of the ACM, Vol.17, No.6, (June, 1974), pp. 337-345.

[10] ----- , Proceedings of the 4th Annual Symposium on Computer Architecture, (March, 1976).

location	content
I+n-1	PI call word from P_{n-1}
.	.
.	.
.	.
I	PI call word from P_0
J+n-1	Program counter save space for preemption of P_{n-1}
.	.
.	.
.	.
J	Program counter save space for preemption of P_0
K+n-1	Program counter save space for signal to P_{n-1}
.	.
.	.
.	.
K	Program counter save space for signal to P_0
L+4mn-4	Descriptor for P_{mn} (see Figure 2)
L+4mn-8	Descriptor for P_{m-1n} (see Figure 2)
.	.
.	.
.	.
L	Descriptor for P_{00} (see Figure 2)
.	
.	Queue Space (see Figure 3)
.	
Q	

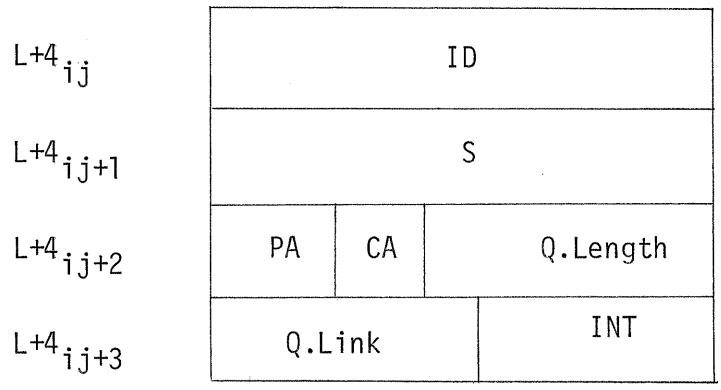
MEMORY ADDRESS RESERVATION

TABLE 1



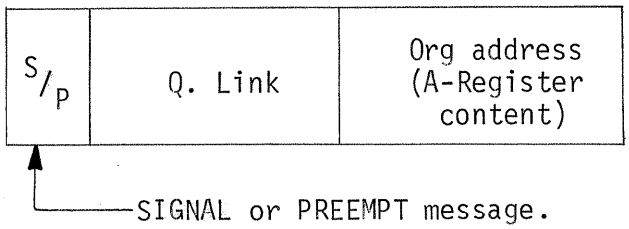
The Multiple Processor Organization

Figure 1



Partial Process Description

Figure 2



Message Queue Entry

Figue 3