HARDWARE SUPPORT FOR INTER-PROCESS COMMUNICATION AND PROCESSOR SHARING*

W.S. Ford
and
V.C. Hamacher

Departments of Electrical Engineering and Computer Science
University of Toronto
Toronto, Ontario, Canada

Abstract

The abstraction of a computer system as a set of asynchronous communicating processes is an important system concept. This paper indicates how the concept could be supported at a low hardware level. A new inter-process communication mechanism called a mailbox is introduced. Examples of its use as a programming tool are given. This is followed by a description of hardware features that use this mechanism as the basis of communication between the components of a complete system. These features include processor-sharing hardware capable of handling process selection and switching with high efficiency. It is also indicated how these features can take the place of conventional input/output structures.

1. Introduction

The abstraction of a computer system as a set of asynchronous communicating processes [1] is a concept which is widely accepted and used by present-day programmers. In fact, this abstraction is now recognized to the extent that most modern systems support it at either a low-level software (e.g., [2,3]) or firmware (e.g., [4,5]) level. Because it is becoming increasingly economical to implement widely-used programming features in hardware, consideration should be given to low-level hardware support for this important abstraction. Much can be gained by going to the basic hardware level, as this removes the restriction of simply implementing sequential algorithms, as is the case when firmware or software features are superimposed upon a conventional hardware structure. We shall therefore propose new low-level features which would exist, for example, at the machine instruction level of a minicomputer, or, in a larger machine, at a level below firmware which implements more complex functions. These features consist of the following:

a) Mechanisms for controlling inter-process communication and synchronization. These facilities should provide a programmer with operators capable of efficiently emulating familiar programming language synchronization primitives (e.g., semaphores [6], monitors [7]) and solving common synchronization problems. They should not need to differentiate between software processes running internal to a CPU and external processes based on peripheral devices or other CPU's. This generality will allow input/output handling and multiprocessing to be incorporated in a natural way.

b) Facilities for handling the sharing of a CPU among a number of logical processes. Process selection and switching should be automatic and fast; however, there should be sufficient flexibility for scheduling policies to be determined under program control.

* This research was supported in part by grant A-5192 from the National Research Council of Canada.

The combination of these features should relieve the programmer from the well-known problems associated with the conventional interrupt mechanism. It is essential that time overheads, especially in process switching, be kept low in order to compete with more conventional input/output structures. An attempt will be made to restrict these proposals to features whose implementation in current or foreseeable technology would be cost-effective, even for application in machines down to the minicomputer scale.

This paper will concentrate on the hardware aspects of these features, and only brief examples of their role in programming will be given. The programming aspects will be discussed further in a future publication. For clarity and consistency, Pascal notation [8] is used throughout for the description of all hardware and software concepts and algorithms.

2. The Mailbox Mechanism

The concept of a low-level mailbox mechanism has been discussed by Spier [9]. He defined the basic characteristics of any interprocess communication mechanism, then introduced a single bit "mailbox" as the "most elementary communication mechanism which would satisfy all of the requirements". His mailbox was capable of transmitting, after initialization, one one-bit message from a sender process to a receiver process. We add practicality to this mechanism with the following two extensions:

a) It is made reusable so that it can pass a stream of messages.

b) A data-carrying capability is added so that a message can convey a word of information.

A mailbox can now be considered as possessing both a state (*full* or *empty*) and contents. In Pascal notation, *mailbox* can be defined as a structured type:

```
type mailbox = record
                 state:  (empty, full);
                 contents:  word
               end
```

The sending and receiving operations on mailbox *m* can be described respectively as *PUT value AT m* interpreted as:

```
repeat until m.state = empty;
m.state := full;
m.contents := value
```

and *GET value AT m* interpreted as:

```
repeat until m.state = full;
m.state := empty;
value := m.contents
```

It must be stipulated that, following a successful until test, the mailbox should be inaccessable to other processes until the assignment operations are complete. This mechanism guarantees that every message sent by a *PUT* operation will be received by exactly one *GET* operation.

This mailbox has considerable value as a low-level programming tool for general synchronization

problems. We present here two brief examples of the use of the mechanism in programming-- the implementation of semaphores and queues.

2.1 Semaphores

The binary semaphore is an important tool as it has been used widely in published solutions to many interesting synchronization problems. It has also been shown [7] to be a suitable mechanism for implementing monitors. Wirth [10] pointed out that a general semaphore corresponds to a message queue which passes only null messages. A similar concept is used here to implement a binary semaphore using a single mailbox. We consider *semaphore* as a type:

```
type semaphore = mailbox
```

and define the operations on semaphore *s* as *P(s)*, implemented as *PUT AT s*, and *V(s)*, implemented as *GET AT s*. The null *value* arguments of the *PUT* and *GET* instructions indicate that the mailbox *contents* field is unused.

With this implementation, the mailbox *full* state corresponds to a semaphore value <1, while the mailbox *empty* state (the natural initial state) corresponds to a semaphore value of 1 (the natural initial state for a mutual exclusion semaphore). A small inconsistency in this implementation is that, strictly, it should be illegal to attempt to execute a *V*-operation on a binary semaphore with value 1. Rather than detect such an attempt as an error, the mailbox mechanism will cause the violating process to be delayed until the next *P*-operation. If the semaphores are used correctly (as in compiler-generated code), the inconsistency will not arise.

2.2 Queues

The mailbox mechanism also provides for a simple implementation of FIFO queues of known maximum length, as required for buffering message streams between processes. This queueing problem has also been referred to as a bounded buffer producer/consumer problem [6,7].

A queue of maximum length *k* words is implemented as an array of *k* mailboxes, all initially *empty*. An in-pointer and out-pointer initially point to one of these locations. A queue of maximum length *k* can therefore be described as the structured type:

```
type queue [k] = record
                   store:  array [1..k] of
                                 mailbox;
                   in, out:  1..k
                 end
```

For a queue *q* two operations are defined. The operation for a producer process to append a word to the tail of the queue is *APPEND word TO q* which can be implemented as:

```
with q do begin
            PUT word AT store [in];
            in := if in < k then in + 1
                  else 1
          end
```

The corresponding operation for a consumer process to remove a word from the head of the queue is *REMOVE word FROM q* implemented as:

```
with q do begin
            GET word AT store [out];
            out := if out < k then out + 1
                   else 1
          end
```

If there is only one producer process and one consumer process operating on the same queue, the above will be correct without any need for semaphores or indivisible operations. Whenever the consumer process attempts to remove a word from an empty queue it will automatically be blocked until the queue is no longer empty. Conversely, if the producer process attempts to append a word when all *k* locations are full, it will be blocked until the consumer removes a word. In the case of more than one producer process for the same queue, it is necessary to enclose the *APPEND* code in a critical region guaranteeing mutual exclusion between producers. A similar modification applies to the case of more than one consumer.

3. Mailbox Memory

We now present a proposal for a hardware implementation of mailboxes which enables them to be used as the basis of communication between the components of a complete system. For this purpose, a physical processor is considered to be any CPU, or any hardware device which logically communicates directly with a CPU process. This includes input/output channels and some peripheral devices. In general, a physical processor may be capable of supporting more than one logical process. The principal path of communication between physical processors is mailbox memory (an array of mailboxes). A possible system configuration is illustrated in Figure 1. This shows all physical processors connected to a common bus which is managed by a mailbox memory controller. Firstly, the mailbox memory and the controller will be described, assuming that each physical processor supports only one logical process. The following section will discuss compatible hardware features for efficiently handling the sharing of a physical processor.

Mailbox memory consists of a number of addressable locations, with word size typically one or two bytes. Any mailbox location may be in either a *full* condition, in which case its contents represent some meaningful value, or an *empty* condition in which the contents are undefined and inaccessable. An additional bit for each word indicates a *full* or *empty* state. Since these state bits are accessed more frequently than the complete words, they can profitably be retained in separate higher-speed memory devices.

The mailbox memory controller receives *PUT* and *GET* requests on the bus, each request specifying a single mailbox memory address. These requests originate from either explicit CPU instructions, or from device interfaces. All are handled identically. The *PUT* operation applied to an *empty* mailbox causes a value to be passed from the processor and stored in the location, the state of that location then becoming *full*. Conversely, a *GET* operation on a *full* mailbox causes the contents of that location to be passed to the processor and the location assumes the *empty* state. The read operation on mailbox contents is allowed to be destructive.

Any attempt to execute a *PUT* operation on a *full* mailbox, or a *GET* operation on an *empty* mailbox causes a BLOCK signal to be sent back to that processor. That processor then enters a mode where it monitors the bus waiting for a WAKEUP signal for that particular mailbox. Whenever a *PUT* or *GET* operation is successfully executed, a WAKEUP signal is broadcast on the bus together with the address of the mailbox involved. When a blocked processor eventually detects the appropriate WAKEUP signal, it then reissues the original *PUT* or *GET* request.

The activity of the mailbox memory controller in processing *PUT* and *GET* bus requests can be described as the following indivisible sequence:

```
while true do
begin
   receive op for mailbox m from processor p;
   state := m.state {Retrieve the state bit};
   if (op = GET) ∧ (state = full) then
```

```
begin  {Successful GET}
   generate WAKEUP (m);
   m.state := empty;
   value := m.contents;
   transfer value to processor p
end
else if (op = PUT) ∧ (state = empty) then
begin  {Successful PUT}
   generate WAKEUP (m);
   m.state := full;
   transfer value from processor p;
   m.contents := value
end
else generate BLOCK (p)  {Unsuccessful
                          PUT or GET}
end
```

An essential feature of the mailbox memory controller is that it can be processing only one *PUT* or *GET* request at any time. Hence there may be times when more than one of the processors are competing for access to the controller. It will be assumed that such competition is resolved on a priority basis, as with conventional bus conflict resolution.

4. Processor Sharing

When the physical processor is a CPU, special provision must often be made for sharing it among a number of internal processes, each being an instance of execution of a machine program. These internal processes must be able to communicate with each other, as well as with external processes, via mailbox memory.

4.1 Process Status Table

Assume that a physical processor may support a maximum of N internal processes. Each such process is assigned a unique identifying number in the range [0, N-1], this number being an index into a hardware process status table. The organization of this table is shown in Figure 2. Naturally, only one process is executing machine instructions at any time, and the identifier of that process is held in a hardware register denoted the current process register. For every process, the status table contains a bit to indicate ready/blocked status, and a register containing a priority value. These entries are used by a despatching mechanism which, when enabled, loads into the current process register the identifier of a ready process selected according to priorities. This despatcher will be discussed in more detail later.

Each internal process is assumed to have its own partition of memory for storage of local data. Each process also has its own set of CPU registers, including program counter and memory bounds registers. The register sets for all processes can be implemented in a high-speed random access memory configuration. When accessing this memory, the most significant portion of the address is obtained from the current process register, so context switching between processes simply involves changing the contents of that register. Therefore, the only time overhead in process switching can be the despatcher delay, which will be discussed later. To briefly support the feasibility of this feature, it should be pointed out that the value of N for a 16-bit minicomputer could reasonably be of the order of 16. With 8 CPU registers this would require a high-speed random access memory of 128 16-bit words. This is not an unreasonably expensive item in current high-speed logic technology.

The execution of a *PUT* or *GET* machine instruction causes the processor to issue an appropriate *PUT* or *GET* request to the mailbox memory controller. If the request can be satisfied directly, the appropriate data transfer is made and execution of the same program continues. If, however, the request cannot be satisfied and the BLOCK signal is returned, the status of the current process is set to blocked and the program counter is not incremented. In each process status table entry there is an additional word of sufficient length to hold a mailbox address. When a process is blocked, the address of the mailbox at which it is blocked is entered in that word. After blocking of a process the despatcher is invoked, causing the processor to switch to a new process.

To handle WAKEUP signals, the processor has an asynchronous mechanism which continually monitors the bus for any WAKEUP signal. On every such signal, this mechanism searches the process status table for processes which are blocked at that particular mailbox, and if any are found their status bits are set to ready. Also, a processor flag is set, indicating that the despatcher should be invoked at the first opportunity. For fast execution of the wakeup phase, it is apparent that the "blocking mailbox" words of the process status table could be configured as an associative memory. For a system with N = 16 and a maximum of 1024 mailbox words, this would call for a relatively inexpensive 16-bit by 10-bit associative memory.

The "wakeup" time can actually be as long as one mailbox contents access time with zero effective time cost. This is because every WAKEUP signal is followed by a mailbox contents access. If the *PUT* or *GET* operation involved was initiated by this processor, then the processor will be delayed anyway until the completion of that access. If the operation was initiated by some other processor, then the "wakeup" can completely overlap program execution. It is impossible for the processor to commence executing a *PUT* or *GET* instruction until the previous mailbox operation is completed, so there is no possibility of "block" and "wakeup" modifications of the process status table clashing, provided:

a) the time for the table adjustment following a WAKEUP signal is less than a mailbox contents access time, and

b) the time for the table adjustment following a BLOCK signal is less than a mailbox state bit access time.

It is clear that a CPU must have special process management instructions for initiating, terminating, and supervising internal processes. There must at least be instructions to enable a process to modify the register contents of another process (for initializing the program counter and memory limits), to set or clear ready/blocked flags, to suspend another process by forcing it to block at a dummy mailbox, and to initialize mailbox states to *empty*. Access to these instructions should be restricted to nominated supervisory processes. There is also a need for instructions to change process priorities. To maintain flexibility in scheduling, it appears that access to these instructions should be relatively unrestricted.

4.2 Despatcher

The purpose of the despatcher is to examine all priority registers and ready/blocked flags of a processor, and load into the current process register the identifier of some ready process whose priority is no lower than that of any other ready process. If no process is ready, it must generate the signal CPUIDLE which inhibits processing. The despatching activity need only be carried out after a change has been made to the process status table, i.e., after a recognized WAKEUP signal, a BLOCK signal, a change priority instruction, or a process management instruction. A synchronization problem arises as the recognition of WAKEUP signals is not synchronized to the basic processor cycle, as the other activities are. In resolving this problem, it should be stipulated that, when the mailbox memory controller broadcasts a WAKEUP

signal, it should not have to wait for responses from processors (i.e., a processor simply "absorbs" a WAKEUP signal). Nor should the speed of the mailbox memory be severely restricted by possible excessive delays on the part of processors in reacting to wakeup signals. For this reason, it appears sensible to synchronize despatching to the processor and provide the despatcher with storage to take a "snapshot" of the ready/blocked flag status whenever it is invoked. On receipt of any WAKEUP signal, the processor wakeup mechanism then has only to execute an associative search and set any required ready flags. It is then capable of immediately accepting another WAKEUP signal. When the despatcher is about to commence its cycle, it latches in the current values of the flags and uses the latched values. Further WAKEUP signals can then be accepted while the despatcher is operating, although any process made ready by such a signal cannot run until after the subsequent despatcher cycle.

The next consideration is the possibility of having despatching overlapping normal processing. Assume that the despatcher is invoked at the end of any instruction cycle in which the process status table was modified, either by the instruction itself (which may have caused a BLOCK, changed priorities, set or cleared ready/blocked flags, etc.) or by the recognition of WAKEUP signals during that period. In the simplest implementation (no overlap), a new instruction cycle is not commenced until the despatcher cycle completes. All despatching time therefore becomes processing overhead. Another approach is to permit processing of new instructions to continue while the despatcher is still operating. For obvious correctness reasons this cannot be done following a BLOCK signal or some process management instructions, but it may be possible to delay process switching if the only table changes that have been made since the last despatching cycle have been caused by priority changes or WAKEUP signals. If this is done, the effects of priority changes and wakeups will be delayed for a limited number of instructions. This will not normally affect correctness.

There exist a variety of possible methods for implementing the despatcher, e.g., sorting networks [11], associative memory [12], or microcoded sequential algorithms. To estimate the achievable speed of a despatcher, consider a simple combinatorial network implementing the required function. This is basically an N-way p-bit digital comparator, where p is the number of bits of each priority register. A fast practical configuration is a tree structure of two-way comparator elements as illustrated in Figure 3. The function of each comparator element is to compare two input priority values, and output both the higher of the values and the identifier of the process having that higher priority. At the lowest level, the priority lines must be gated with the corresponding ready flags to ensure that only processes with ready status are considered (assume priority 0 is equivalent to blocked status).

Assuming N processes, the delay time for this circuit will clearly be $\lceil \log_2 N \rceil \tau$, where τ is the delay of a p-bit comparator. This comparator is equivalent to a p-bit subtractor, and it can be shown that an achievable delay for such a circuit is $2 + 2\lceil \log_F p \rceil$, where F is the maximum permissible fan-in. A large range of priority values may be desirable for implementing, for example, several of Hoare's algorithms [7]. Assuming p up to 16, F = 4, and a gate delay of 20 nsec., this would give τ = 120 nsec. For a processor supporting 16 processes, the total despatcher delay would be 480 nsec., i.e., of the same order as a memory cycle time.

4.3 Process Modules

An important system parameter is N, the maximum number of processes supported by a CPU. It is essential that N be sufficiently large for any given application, but not be excessively large because of the high cost in wasted register sets, associative memory, etc. It should therefore be a highly flexible parameter. One way of achieving this is the use of a modular hardware structure, where each of m modules contains the registers, status table, associative memory, and section of the despatcher for n processes, where N = mn. Any particular machine can then be built with as many modules as required for the particular application, and machine capability can be expanded as required by adding modules.

With this modular approach, the tree structure model of the despatcher (Figure 3) is no longer acceptable. To estimate achievable despatcher speed, consider, therefore, an array structure as shown in Figure 4. In this array, each A element is a 2-way p-bit comparator, and each B element is an n-way p-bit comparator. The time delay in an A element will be τ, and in a B element will be $\lceil \log_2 n \rceil \tau$. The total despatcher delay will therefore be the basic delay in B elements plus the time for the result to propagate through the A elements, i.e., $(\lceil \log_2 n \rceil + m)\tau$. Hence, for speed reasons, n should be large relative to m; however, for flexibility n should be small. In a realistic situation n might be 4 or 8. (It is, in fact, possible to combine modules of different n in the same system.)

Assuming a value of n = 4 and all other parameters the same as for the despatcher discussed previously, the total delay for a modular despatcher would be 720 nsec. which is still an acceptable value.

5. Input/Output

In conclusion, we shall demonstrate the role of the proposed hardware features in the driving of conventional input/output devices.

Consider, firstly, the class of devices whose basic unit of data transfer is no more than a few bytes. The class includes keyboards, teleprinters, paper tape equipment, real-time clocks, process control interfaces, etc. As was shown in Figure 1, these devices can be configured so as to communicate directly with the mailbox memory controller by *PUT* and *GET* bus requests. From a system point of view each device can therefore be considered to be executing an internal program containing *PUT* and/or *GET* statements.

For example, an output device such as a teleprinter may be considered to be executing the program:

```
while true do
begin
   GET characater AT teleprintout;
   print character
end
```

where *teleprintout* is a mailbox dedicated to that device. A CPU process can then send a character to the teleprinter by executing the single instruction *PUT character AT teleprintout*. Output to the teleprinter could be buffered using the FIFO queue mechanism as follows. Assume a queue *printqueue* of sufficient maximum length, then the main process code for emitting a character is *APPEND character TO printqueue*. An additional CPU buffer process executes the following program:

```
while true do
begin
   REMOVE nextchar FROM printqueue;
   PUT nextchar AT teleprintout
end
```

This process effectively takes the place of a conventional device interrupt service routine. For efficient device operation it should, of course, have a relatively high priority.

Input devices can be handled similarly. For example, a keyboard may be considered as executing the following program:

```
while true do
begin
   receive character from operator;
   PUT character AT keyboardin
end
```

A CPU process then receives a character from the keyboard by executing the instruction *GET character AT keyboardin*. It is assumed that if the device is in its blocked internal state it is incapable of accepting another character from the operator, e.g., the keyboard is locked. Again it is possible to buffer the device using the FIFO queue mechanism and a dedicated CPU buffering process.

Special consideration must be given to devices such as disks which require high-speed transfers of large blocks of data to or from conventional memory. With these transfers it would be unrealistic to pass all data through mailbox memory, so we assume the existence of some form of channel which controls the direct transfer of blocks of data between the device and conventional memory. However, certain communications between the channel and CPU processes will be passed via mailbox memory. These include requests for transfers, notification of completion of transfers, and notification to the CPU of any error conditions. A CPU process initiates a transfer by depositing an appropriate message in a dedicated mailbox known to the channel. Several processes can thereby share the device with conflicts being automatically resolved on a priority basis at that mailbox. To wait for the completion of its transfer, a process waits for a response from the channel at another dedicated mailbox. To handle error conditions, a convenient approach is to have channels and devices report all error conditions to special CPU processes dedicated to handling such conditions, rather than report them to the process requesting the transfer. Each special process waits at a mailbox for notification of an error and can take any required action (e.g., notify the operator). It then responds to the channel that it should either repeat or abort the transfer. It is possible to share the same error-handling process among a number of channels and/or devices. This provides a very convenient way for handling similar error conditions at different devices; for example, all console display messages regarding device states can now originate in the one process.

6. References

1. Horning, J.J. and Randell, B. (1973), Process Structuring, Computing Surveys, Vol. 5, No. 1, pp. 5-30.
2. Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. (1974), HYDRA: The Kernel of a Multiprocessor Operating System, Comm. ACM, Vol. 17, No. 6, pp. 337-345.
3. Ritchie, D.M. and Thompson, K. (1974), The UNIX Time-sharing System, Comm. ACM, Vol. 17, No. 7, pp. 365-375.
4. Liskov, B.H. (1972), The Design of the Venus Operating System, Comm. ACM, Vol. 15, No. 3, pp. 144-149.
5. Atkinson, T. (1974), Architecture of Series 60/ Level 64, Honeywell Computer Journal, Vol. 8, No. 2, pp. 94-106.
6. Dijkstra, E.W. (1968), Cooperating Sequential Processes, in Programming Languages, F. Genuys (ed.), Academic Press, New York, pp. 43-112.
7. Hoare, C.A.R. (1974), Monitors: An Operating System Structuring Concept, Comm. ACM, Vol. 17, No. 10, pp. 549-557.
8. Wirth, N. (1971), The Programming Language Pascal, Acta Informatica 1, pp. 35-63.
9. Spier, M.J. (1973), Process Communication Prerequisites or the IPC-Setup Revisited, 1973 Sagamore Conference on Parallel Processing, Syracuse University, pp. 79-88.
10. Wirth, N. (1969), On Multiprogramming, Machine Coding, and Computer Organization, Comm. ACM, Vol. 12, No. 9, pp. 489-498.
11. Thurber, K.J. (1974), Interconnection Networks - A Survey and Assessment, National Computer Conference, Vol. 43, pp. 909-919.
12. Berg, R.O. and Johnson, M.D. (1970), An Associative Memory for Executive Control Functions in an Advanced Avionics Computer System, Proc. 1970 IEEE International Computer Group Conference, pp. 336-342.

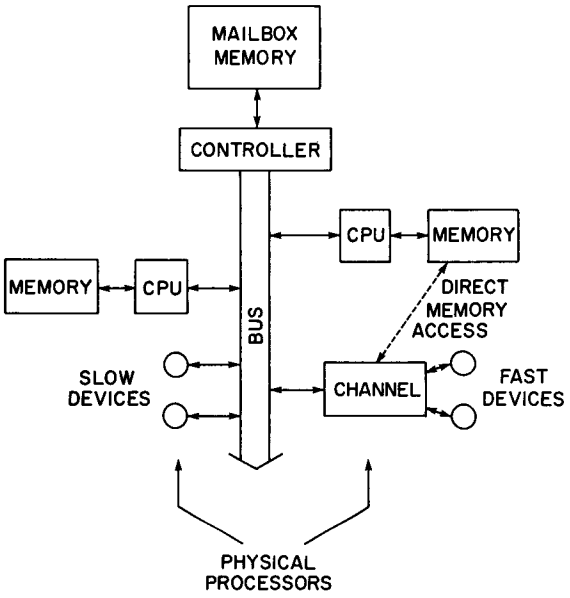

Figure 1: Possible System Configuration

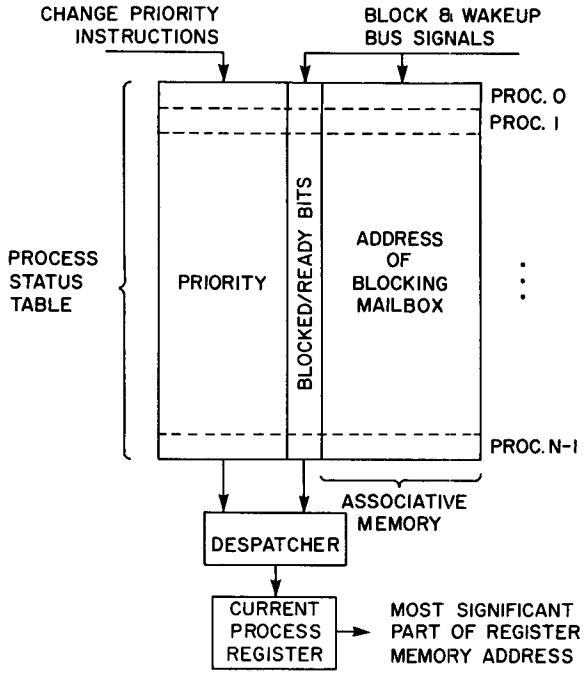

Figure 2: Processor-Sharing Hardware

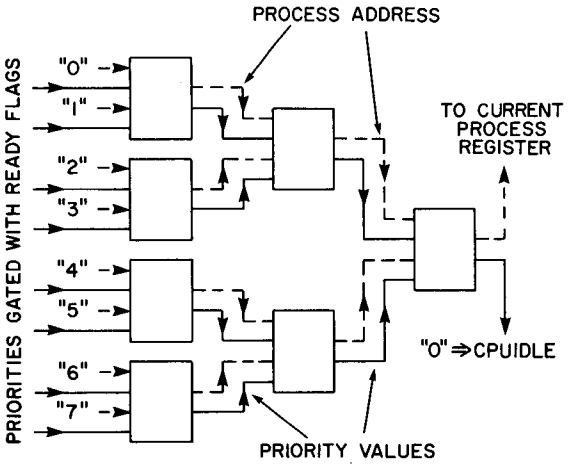

Figure 3: Combinatorial Despatcher for 8 Processes

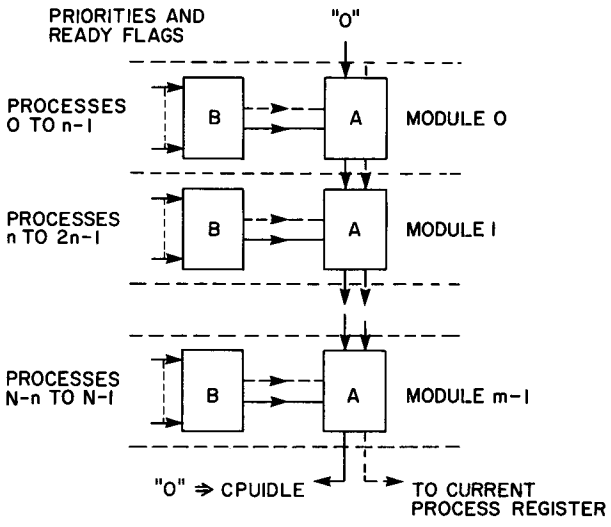

Figure 4: Structure of a Modular Despatcher