Low level architecture features for supporting process communication

W. S. Ford and V. C. Hamacher
Departments of Electrical Engineering and Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4

A proposal is presented for low level hardware features which would assist in the realisation of the abstraction of a computer system as a set of asynchronous communicating processes. A low level synchronisation and communication mechanism, called a mailbox, is described, together with details of a hardware structure for configuring a complete system around a set of these mailboxes. Programming for this architecture is then discussed. It is shown how the new features can be used for controlling input/output, and for handling general synchronisation.

(Received June 1975)

1. Introduction

The purpose of this paper is to propose computer design features to assist in the realisation of the abstraction of a computer system as a set of asynchronous communicating processes (Hornig and Randell, 1973). These proposals are motivated by the premise that it is not completely adequate merely to superimpose firmware or low level software features upon a conventional hardware structure to realise this abstraction, which is now accepted as a fundamental programming concept. The following characteristics are considered to be essential:

(a) Communication primitives 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. It should also eliminate the need for the conventional interrupt mechanism.

(b) The functions of synchronisation and processor scheduling should be clearly separated. Synchronisation applies to all processes of a system. Its purpose is to retain system correctness regardless of rates of progress of processes. Scheduling applies only to processes which share a physical processor, and its purpose is to resolve conflicts for use of the processor. The aim here is to superimpose highly flexible scheduling capabilities upon a more rigid and controlled synchronisation protocol.

(c) The low level hardware communication primitives should be capable of efficiently emulating familiar programming language synchronisation primitives, e.g. semaphores (Dijkstra, 1968a), and monitors (Hoare, 1974).

(d) In an implementation in current or foreseeable technology, time overheads should be sufficiently low for these features to be competitive with conventional input/output structures.

It is envisaged that these mechanisms would exist at the machine instruction level of a minicomputer system, or alternatively, in a larger machine, at a level below firmware which implements more sophisticated primitives. Aspects such as protection between noncooperating processes will therefore be given little attention, it being assumed that responsibility for those considerations will be shared by programmer, programming language, and firmware features.

This paper will firstly introduce a synchronisation and communication mechanism particularly suited to the above requirements. An outline is then given of hardware features that use this mechanism as the basis of communication between the components of a complete system. The remainder of the paper is devoted to techniques for programming such a hardware structure, and includes basic programming tools, and examples of the driving of conventional input/output devices. For clarity and consistency, Pascal notation (Wirth, 1971) is used throughout for the description of all hardware and software concepts.

2. The mailbox communication mechanism

Before specifying hardware mechanisms for controlling interprocess communication, we will look briefly at existing high level communication primitives and see how the hardware primitives will relate to them. The high level primitives have all been proposed at a programming language level for solving particular programming problems. There is no reason why the hardware primitives should correspond exactly to any one of them; rather, the hardware primitives must reflect the elements of interprocess communication, while being simple to implement efficiently. They should also lend themselves to simple and efficient solutions to the problems for which the high level primitives were created.

The high level primitives referred to can in general be classified into one of the two following categories:

(a) pure synchronisation primitives
(b) data transfer primitives (with implicit synchronisation).

In the first category, no data transfer capability is associated with the primitives. Included in this class are block/wakeup (Saltzer, 1966), semaphores, extended semaphores (Vantilborgh and van Lamswaerde, 1972; Wodon, 1972), conditional critical regions (Hoare, 1972; Brinch Hansen, 1973), and condition variables (Hoare, 1974). Only the simpler of these appear suitable for efficient implementation at a low level, e.g. semaphores have been implemented in different operating systems (Dijkstra, 1968b; Liskov, 1972; Wulf et al., 1974). With primitives in the second category, data values (usually considered to be messages) are actually passed between processes via the primitives. This category includes communicating semaphores (Saal and Riddle, 1971) and numerous other message passing schemes, many of which have been implemented in software (Brinch Hansen, 1970; Ritchie and Thompson, 1974; Stoy and Strachey, 1972; Morris et al., 1972) or firmware (Atkinson, 1974). One reason why these schemes are so attractive is the clear way in which they relate to the familiar concept of stream input/output. Data transfer primitives are generally not trivial to implement cleanly in hardware, mainly because of difficulties in implementing FIFO queues.

In proposals for pure synchronisation primitives, much consideration has always been given to the solution of problems of a producer/consumer nature—problems which are solved trivially by data transfer primitives. Because these problems

*This research was partially supported by grant A-5192 from the National Research Council of Canada. The material forms part of the doctoral dissertation research of W. S. Ford.
Occur so naturally in computer systems, it is suggested that the concept of controlled data transfer between processes is so basic to interprocess communication that it should be supported by basic hardware structures, rather than be considered as a programming problem to be solved using synchronisation primitives. On the other hand, data transfer primitives tend to be too unwieldy for the solution of synchronisation problems not involving data transfers. Hence, if the number of separate concepts is to be kept small, a primary aim in selecting hardware primitives must be to attain some measure of data transfer capability without this capability being a burden when dealing with pure synchronisation problems.

A useful starting point is the low level approach to the interprocess communication problem taken by Spier (1973). Having defined the characteristics of any interprocess communication mechanism, he then defines the 'most elementary communication mechanism which would satisfy all of the requirements'. This mechanism is a single bit 'primitive mailbox' which is capable of transmitting one bit message from a sender process to a receiver process. It can be considered as a Boolean variable b initialised to the false state. The two operations on the mechanism are then a sending operation of the form:

\[ b := \text{true} \]

and a receiving operation of the form:

\[ \text{repeat until } b; \]

This mechanism is indeed elementary in terms of conventional logic elements, but is of little practical use. The principal problem is that it can be used to pass only one message after initialisation. A simple modification is to make this primitive mailbox reusable by simply initialising it to the false state, but making the sending operation of the form:

\[ \text{repeat until } \neg b; \]

\[ b := \text{true} \]

and the receiving operation of the form:

\[ \text{repeat until } b; \]

\[ b := \text{false} \]

With both operations it is also stipulated that between a successful until test and the subsequent assignment operation, the mechanism must be inaccessible by any other process. These symmetric operations clearly provide the capability for sending processes to transmit a stream of one bit messages to receiving processes, with the latter indicating their readiness to accept messages, all via this one bit mailbox. Such a mechanism can in fact be found in existing systems; for example, the manipulation of a device busy/done flipflop during program controlled input/output in a DEC PDP-8.

The one bit mailbox is still restricted, as was Spier's mechanism, in that the only information that can be conveyed by a message is its own existence. To provide the desired data transfer capability, a logical extension is to associate a one bit mailbox with another variable (e.g. a memory word), with reading or writing of the value of this variable being side effects of operations on the one bit mailbox. More specifically, when a process sends a message, it passes a value which is stored in an associated memory location at the instant the control bit switches from false to true. Conversely, in a receive operation, the contents of the memory location are read and passed to the receiving process when the control bit switches from true to false. The one bit mailbox plus its associated memory location will now be referred to simply as a mailbox, possessing both a state (full or empty) and contents. In Pascal notation, mailbox can be considered as a structured type defined as:

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

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

\[ \text{repeat until } m.\text{state} = \text{empty}; \]
\[ m.\text{state} := \text{full}; \]
\[ m.\text{contents} := \text{value} \]

and GET value AT \( m \) interpreted as:

\[ \text{repeat until } m.\text{state} = \text{full}; \]
\[ m.\text{state} := \text{empty}; \]
\[ \text{value} := m.\text{contents} \]

Again it must be stipulated that, following a successful until test, the mailbox should be inaccessible 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.

3. A proposed hardware structure

A proposal will now be presented for implementing mailboxes in hardware, and using them 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 whichlogically communicatesdirectly 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 Fig. 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. This will be followed by a brief outline of compatible hardware features for efficiently handling the sharing of a physical processor. These hardware features and their time overheads are discussed in more detail elsewhere (Ford and Hamacher, 1976).

3.1. Mailbox memory

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 inaccessible. 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 the requesting 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
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

Since this mailbox memory controller can be processing only one PUT or GET request at any time, 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.

3.2. A multiprocessor
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.

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 organisation of this table is shown in Fig. 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/blacked status, and a register containing a priority value. These entries are used by a dispatching mechanism which, when enabled, loads into the current process register the identifier of that process which has ready status, and which has the highest priority value of all ready processes. If more than one of the ready processes have equal highest priority, the dispatcher must select only one of them, for example, that with the smallest identifier number.

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 involves simply changing the contents of that register.

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 dispatcher 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 at the first opportunity the dispatcher should be invoked. For fast execution of this wakeup phase, it is apparent that the 'blocking mailbox' words of the process status table could be configured as an associative memory.

A particularly important parameter of the multiprocessor processor is the value of $N$, the maximum number of internal processes supported by a CPU. It is essential that $N$ be sufficiently large for any application, but not be excessively large because of the high cost of wasted register sets, associative memory, etc. One way of achieving flexibility would be the provision of hardware modules where each module contains the registers, status table, associative memory, and section of the dispatcher for a small number of processes. Any particular machine could then be built with as many modules as required for the particular application. It would also be possible to expand a machine capability as required by adding modules.

### 3.3. Process management instructions

It is clear that a CPU must have special 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 initialising the program counter and memory limits), to set or clear readyblocked flags, to suspend another process by forcing it to block at a dummy mailbox, and to initialise mailbox states to empty. Because of the power of these instructions, and the infrequency of their use, it would appear desirable to restrict their usage to processes running in a privileged mode, even in small dedicated machines. It is probably sufficient to permit only one particular process (say process 0 in any CPU) to have access to these privileged instructions, and in any system dedicate that process to management tasks. That process would normally be dormant awaiting special requests from other processes, e.g. create, destroy, suspend, or debug functions. The management process could check the validity of any request before honouring it, and could maintain any desired structure such as a process creation hierarchy.

There is also a need for instructions to manipulate process priorities. To maintain flexibility in scheduling, access to these instructions should be relatively unrestricted. Erroneous use of these instructions is not as serious as with the other management instructions, as it can only lead to observable timing discrepancies.

### 4. Some basic programming techniques

Programming for this architecture can now be considered. This section presents four programming tools; semaphores, compound mailboxes, queues, and shared variables. It is envisaged that even the lowest level programming language would provide primitive operators to support these. It will be shown how each can be implemented using the hardware mailbox memory facilities.

#### 4.1. Semaphores

The binary semaphore is an important tool as it has been used widely in published solutions to many interesting synchronisation problems, and is a suitable lower level mechanism for implementing monitors. Wirth (1969) 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. Consider semaphore as a distinct type:

```plaintext
type semaphore = mailbox
```

The operations defined for semaphore $s$ are $P(s)$, implemented as:

```plaintext
PUT AT s
```

and $V(s)$, implemented as:

```plaintext
GET AT s
```

The null value arguments of the PUT and GET instruction 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 these semaphores are used correctly (as in code generated by a compiler from higher level constructs), the inconsistency will not arise.

Notice that it is possible to have any number of processes waiting on the same semaphore. A $V$-operation will always awaken the waiting process that currently has the highest priority. Scheduling of the semaphore queue is therefore moderately flexible, but particularly suited to priority disciplines. This may be of benefit in some situations; for example, it permits obvious simplification of the semaphore solution of the 'second readers and writers problem', (Courtois et al., 1971). It is also valuable for implementing monitors with scheduled waits. The basic monitor implementation using semaphores was described by Hoare (1974). This did not include the scheduled wait facility which enables a process to specify a relative priority whenever it may be forced to block awaiting the signalling of a condition. It is only necessary to precede the $P$-operation (constituting the wait) by an instruction setting the hardware priority of that process to a level depending upon the priority value specified by the program. The $P$-operation is then followed by an instruction returning the hardware priority to its normal level.

### 4.2. Compound mailboxes

For reasons of utilisation efficiency, the physical mailbox word size should be kept small. To preserve programming generality in passing messages of arbitrary size, there is therefore a need for a construct which performs logically as a mailbox, but permits the passing of a message of size greater than the mailbox word size. One such construct with full generality is the compound mailbox. A compound mailbox of width $w$ words is implemented using an array of $w$ simple mailboxes. It can be considered as a new type:

```plaintext
type mailbox [w] = array [1 . . . w] of mailbox
```

The operations that can be performed on compound mailbox $c$ are:

```plaintext
PUT word1 , word2 , . . . , word_w AT c ;
```

and

```plaintext
GET word1, word2, . . . , word_w AT c .
```

The following implementation of these operations is correct

---

*It is actually necessary to awaken only the highest priority process blocked at the mailbox and this technique of awakening all processes may result in some fruitless attempts by lower priority processes to re-execute $P$ and $GET$ operations. However, it would appear difficult to justify the inclusion of hardware to effect a priority search at this point.
for any number of processes attempting to simultaneously perform compound PUT's and GET's:

\[ \text{compound PUT:} \]
\[ \text{for } i := 1 \text{ to } w \text{ do PUT word, AT } c[i]; \]
\[ \text{compound GET:} \]
\[ \text{for } j := w \text{ downto } 1 \text{ do GET word, AT } c[j]; \]

where \( c[i] \) is the simple mailbox constituting element \( i \) of the array.

It is important to note the ordering of the simple mailbox operations. In particular, the first simple PUT operation in a compound PUT operates on \( c[1] \), and the last operation in a compound GET operates on \( c[1] \). The simple mailbox \( c[1] \) thereby acts not only as storage for one word of the message, but also as a semaphore guaranteeing mutually exclusive access to the array for processes executing compound PUT's. For example, suppose processes \( P \) and \( Q \) are both attempting to execute compound PUT's on \( c \) while process \( R \) is executing compound GET's. If process \( P \) succeeds in completing a PUT . . . AT \( c[1] \), then process \( Q \) cannot PUT to the array until process \( R \) executes a GET . . . AT \( c[1] \), i.e. until process \( P \) has completed its entire compound PUT and process \( R \) has completed the subsequent compound GET. In the same way, \( c[w] \) functions as a semaphore guaranteeing mutually exclusive access to the array for processes executing compound GET's. Mailbox \( c[w] \) must be the last simple mailbox operated upon in a compound PUT and the first in a compound GET. The remaining simple mailboxes function only as storage locations, and the relative order in which they are operated upon is of no consequence.

4.3. Queues

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

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

\[
\text{type queue \[k\] = record}
\text{ store: array \[1 . . k\] of mailbox;}
\text{ in, out: } 1 . . k
\text{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:

\[
\text{with } q \text{ do begin}
\text{ PUT word AT store \[in\];}
\text{ in := if in < k then in + 1 else 1}
\text{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:

\[
\text{with } q \text{ do begin}
\text{ GET word AT store \[out\];}
\text{ out := if out < k then out + 1 else 1}
\text{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. Note that this is achieved without the need for retaining a count of items in the queue, and neither process ever has to explicitly check the state of the queue or the other process.

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.

4.4. Shared variables

A useful technique for handling variables shared between processes (e.g. status flags, queue pointers) is to allocate storage for them in mailbox memory. This not only makes them accessible by all processors but also provides a convenient mechanism for building critical regions. Consider shared variable \( v \) which is assigned storage in the (possibly compound) mailbox at address \( M_v \). Any process wishing to access this variable also has a local variable \( v \), and when it wishes to read, and/or modify the shared variable it executes code of the following form:

\[
\text{GET } v \text{ AT } M_v;
\text{ v := f(v);}
\text{PUT v AT } M_v
\]

Any code between the GET instruction which 'fetches' the variable, and the PUT instruction which 'returns' it, constitutes a critical region guaranteeing exclusive access to the shared variable. This can be used to implement the programming construct (Brinch Hansen, 1973):

\[
\text{region v do}
\text{ v := f(v)}
\text{end}
\]

The compiler ensures that reference to variable \( v \) is made only from within such a region.

5. Input/output programming examples

We shall now demonstrate the role of the hardware features and programming tools in the driving of conventional input/output devices.

First consider the class of devices whose basic unit of data transfer is no more than a few bytes. This class includes keyboards, teleprinters, paper tape equipment, real time clocks, process control interfaces, etc. As was shown in Fig. 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:

\[
\text{while true do}
\text{ begin}
\text{ GET character AT teleprintout;}
\text{ print character}
\text{ end}
\]

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. Assume a queue printqueue of sufficient maximum length, then the code for emitting a character is APPEND character TO printqueue. An additional internal buffer process executes the following program:

\[
\text{while true do}
\text{ begin}
\text{ REMOVE nextchar FROM printqueue;}
\text{ PUT nextchar AT teleprintout}
\text{ end}
\]
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 buffering process. In fact, it is even possible to multiplex a number of keyboards through the one keyboardin mailbox, and use a single buffering process. This is achieved by having each keyboard deposit its unique identifier in the mailbox along with each character. The buffering process continually reads from this mailbox and routes the character to the correct queue as determined from the keyboard identifier.

Special consideration must be given to devices 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.

As an example, consider driving a disc. The code for CPU process \( i \) to initiate a transfer and wait for its completion may be:

```
PUT i, command, addresses AT startdisc;
... [processing which may overlap transfer];
GET AT proceed
```

where startdisc is a compound mailbox into which all details of a request are entered by the initiating process, and proceed is a mailbox at which the process will be blocked until it receives explicit notification from the disc channel that the transfer is complete. Hence, it is possible for any number of processes to share the disc, the requests being automatically queued, if necessary, at startdisc, and serviced according to process priorities. The disc channel may be considered as executing internally a program of the following form:

```
while true do
  begin
  GET process id, command, addresses AT startdisc;
  repeat
    execute command using addresses;
    if (no error) then PUT AT proceed [process id]
  else begin
    PUT (error details) AT errormsg;  
    GET (response) AT errorresponse
  end
  until (no error) ∨ (response = abort)
end
```

There are two special points to note about this program. Firstly, notice that the mailbox proceed is actually an element of an array so that each initiating process has its own proceed mailbox. This is essential for correctness, as it is possible for two processes to complete PUT operations on startdisc and be waiting for the signal to proceed. The channel must then be able to differentiate between these processes. This approach also permits the extension to a more intelligent channel (or intervening software process) capable of accepting several requests and reordering them to optimise disc access times.

The second point to note is the handling of error conditions. The general 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. In this case, the error handling process would be programmed as follows:

```
while true do
  begin
    GET (error details) AT errorstatus;
    take required action;
    PUT (response) AT errorresponse
  end
```

This process can take any required action (e.g. notify the operator) and can respond to the channel that it should either repeat or abort the transfer. In the latter case, it must also have the initiating process terminated, otherwise it would be indefinitely delayed at its proceed mailbox. It is possible to share the same error handling process among a number of channels and/or devices. This provides a 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. Conclusion

The primary aim of this paper has been to propose architecture features for supporting the communicating process concept at a low level. To this end, we have discussed new hardware features for handling:

(a) process communication among all components of a system via a set of mailbox communication primitives;
(b) sharing of a physical processor by a number of logical processes.

The programming examples have demonstrated techniques for driving different classes of input/output devices using only these hardware features. These examples have not been exhaustive and it may, in fact, be necessary to extend the scope of the input/output features in exceptional situations. There appears to be no need for any mechanism akin to the familiar interrupt, although this has not been positively established.

The mailbox mechanism has also been shown to be a suitable underlying mechanism for controlling general process synchronisation. Buffered message queues and binary semaphores both find simple and natural implementations. These can in turn be used to implement more complex synchronisation tools, e.g. monitors.

The question of scheduling has not been discussed in detail. The attitude is that the hardware priority structure and the priority manipulation capability provide a high degree of flexibility. Scheduling can be superimposed upon a system in which synchronisation primitives have been used to guarantee correctness regardless of the scheduling strategy.

### Acknowledgement

The authors are indebted to Professor R. C. Holt of the Computer Systems Research Group of the University of Toronto for many helpful comments regarding the material presented herein.

### References

