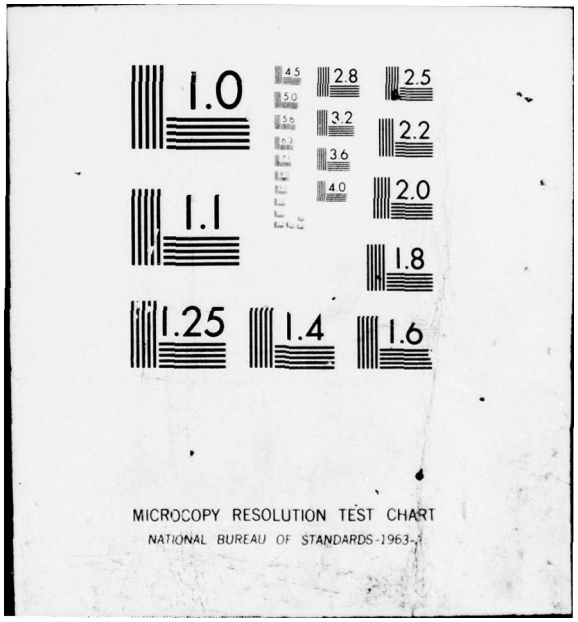
AD-A068 162 UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES DEPT OF--ETC F/G 9/2
MULTIPROCESSOR ARCHITECTURES FOR CONCURRENT PROGRAMS,(U)
APR 78 P B HANSEN N00014-77-C-0714
UNCLASSIFIED NL

END
DATE FILMED
6-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(15) Contract N00014-77-C-0714

LEVEL II

12 B.S.

(6) MULTIPROCESSOR ARCHITECTURES FOR CONCURRENT PROGRAMS,

(10) Per Brinch/Hansen

Computer Science Department
University of Southern California
Los Angeles, California 90007

(12) 21p.

(11) 8 April 1978

Abstract This paper proposes a hierarchical multiprocessor architecture for real-time programs written in a concurrent programming language. The use of processes and monitors leads to a multiprocessor system in which each processor has a local store dedicated to a single process. The processors share a common store that contains the monitors. To avoid congestion in the common store the processes and monitors are partitioned into subsystems that share a hierarchy of common stores. The main goal is to develop a synthesis of an abstract language and a computer architecture that match in an obvious way.

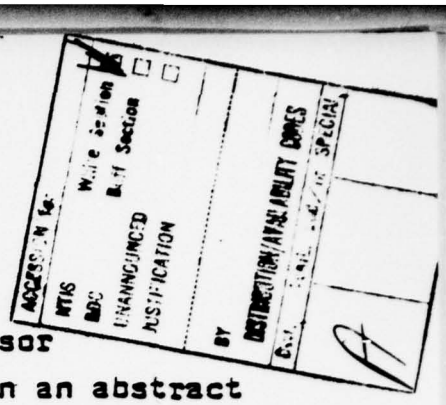
Key Words and Phrases: language-directed computer design, multiprocessor architecture, hierarchical stores, real-time applications, concurrent programming, processes, monitors

CR Categories: 4.2, 4.32, 6.2

Presented at the ACM 78 Conference, Washington, D.C., Dec. 1978.

This work has been supported by the Office of Naval Research under contract NR ~~049-415~~ 048-647

410617



1. Introduction

This paper proposes two hierarchical multiprocessor architectures for real-time applications written in an abstract language for concurrent programming.

The proposal rests on the assumption that simplicity, reliability, and efficiency are essential for real-time applications. Without simplicity one cannot expect to understand the purpose and details of a large program. Without reliability one cannot seriously depend on it. And without efficiency a real-time program cannot keep pace with its environment.

Although efficiency is important we will not let it compromise the vital need for simplicity. Where such a conflict exists, we will settle for a simple system that can handle many (but not all) real-time applications.

These programming goals can be met by careful design of the programming language, the compiler, and the computer architecture:

(1) To obtain simplicity real-time programs must be written in an abstract language that supports modular programming.

(2) To make real-time programs reliable the programming language must permit extensive compilation checks that ensure the integrity of modules.

(3) To make real-time programs efficient they must be executed by a multiprocessor system in which each processor is dedicated to a single process.

(4) To make the language implementation straightforward the multiprocessor architecture must support the language concepts in an obvious way.

(5) To make the multiprocessor inexpensive it must use microprocessor technology.

Initially, we looked at programming languages in which all communication between concurrent processes takes place by messages only. This approach seemed natural for microprocessor networks without common store modules. Message systems with completely deterministic input/output behavior have existed for more than a decade. But a recent proposal has pointed out an occasional need for nondeterministic process communication [1]. Since these ideas are still at a very early stage, we felt that it would be more appropriate for us to experiment in only one area at a time.

Now we do have several years of experience with the programming language Concurrent Pascal that includes processes and monitors [2]. So we decided to look for a multiprocessor architecture that supports a language of this type directly.

In any multiprocessor system, processes need to communicate to cooperate on common tasks. Each process uses some portion of the transmission capacity of the communication lines (which may be bus lines or common stores). As more processes are connected to the same line, a point is soon reached where that line becomes a bottleneck.

So a key problem is to avoid congestion of the communication lines. To do that one must take advantage of the locality of store references within a concurrent program. One may, for example, observe the program during its execution to discover patterns of store references. Demand-paging systems and cache stores are typical examples of this approach. Both require complicated run-time mechanisms.

We will instead depend on the compiler to exploit the locality of references that is determined by the modularity of the programming language. The use of processes and monitors naturally leads to a multiprocessor system in which each processor has a local store dedicated to a single process and in which several processors share a common store that contains the monitors.

Although performance measurements are scarce for concurrent programs preliminary estimates for Concurrent Pascal programs suggest that each process refers to its own code and variables an order of magnitude more often than it refers to the monitors. The simplest architecture proposed here therefore consists of about 10 processors each with their own store and sharing a single common store.

For systems with up to 100 processors we propose a block-structured programming language that maps directly onto a multiprocessor architecture with a hierarchy of common stores.

We would expect that more general multiprocessor systems, such as the Cm*, can be configured to handle the special cases presented here [3]. Our purpose, however, is to take full advantage of the characteristics of a concurrent programming language and develop a synthesis of a language and an architecture that match in an obvious way. This viewpoint leads to many simplifying assumptions that should have a beneficial effect on the cost and reliability of the hardware. It is our view that the search for utter simplicity and full generality are complimentary (but often conflicting) research goals. Both approaches should be tried before any firm conclusions can be made about their merits.

2. Programming concepts

Most complex systems in nature are organized hierarchically as subsystems which in turn may be further subdivided [4]. In such systems, each subsystem spends most of its time performing an autonomous function and uses much less time interacting with a few other subsystems. The exact timing of events in each subsystem is independent of the exact timing of other subsystems. In the short run the behavior of each subsystem is independent of other subsystems. In the long run a subsystem is influenced only by the average behavior of other subsystems.

In the design of a real-time computer system one can take advantage of the hierarchical nature of the world by identifying subtasks that are nearly autonomous and which are loosely connected to one another. The great advantage of such systems is that they can be designed, tested, and tuned piecemeal by focussing the attention on one subsystem at a time.

A real-time application will be controlled by a concurrent program that runs on a multiprocessor system dedicated to that application. The subtasks will be performed by a fixed number of asynchronous processes that are executed simultaneously. The processes communicate by means of a fixed number of monitors [5, 6].

Figure 1 shows an example of two processes that communicate by means of a buffer monitor. The arrows indicate that these processes have access to that monitor.

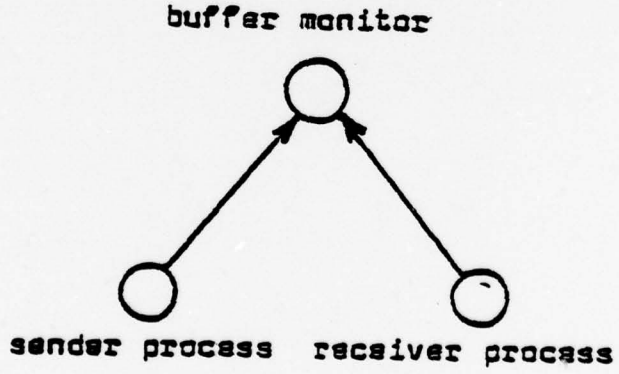


Fig. 1. A hierarchical subsystem

The following shows how the buffer monitor can be programmed:

```
monitor buffer
var slot: message; full: boolean

procedure send(m: message)
when not full: slot:= m; full:= true end

procedure receive(var m: message)
when full: m:= slot; full:= false end

begin full:= false end
```

The monitor defines a common data structure consisting of a message slot and a boolean indicating whether or not it is full. The monitor also defines two operations, send and receive, on the buffer and an initial statement that makes it empty to begin with.

Processes can perform the send and receive operations on the buffer but cannot access the data structure directly. This is guaranteed by the compiler.

The operations on a monitor take place strictly one at a time. When a process performs a monitor operation the computer will delay further operations on the same monitor until the current operation is finished. These short-term delays are implicit in the monitor concept.

A monitor operation may, however, postpone its own completion until the common variables satisfy a certain condition. These medium-term delays are expressed in the language by means of a statement of the form [7]:

```
when boolean expression: statement end
```

This delays the execution of the next statement until a boolean expression becomes true (as the result of another monitor operation).

When a monitor operation delays itself explicitly another monitor operation can take place. The delayed operation is resumed when its precondition is satisfied and no other operation is in progress.

The boolean expressions used for synchronization are more elegant, but less efficient than the queue variables used in Concurrent Pascal [2, 8]. The low cost of microprocessor technology should, however, make the more elegant concept the obvious choice.

The example below shows two concurrent processes that communicate by means of the monitor buffer:

```
process sender
var x: message
cycle produce(x); buffer.send(x) end

process receiver
var y: message
cycle buffer.receive(y); consume(y) end
```

Each process defines a local data structure consisting of a single message and a cyclical statement that operates on it. The variables of one process are inaccessible to other processes. This too is guaranteed by the compiler. The checking of access rights during compilation makes a hardware protection system unnecessary [2].

The processes are loosely connected if they spend most of their time operating on their local data structures and very little time on exchanging data. Loosely coupled processes spend only a fraction of their time within monitors.

A monitor can also be used to control the access to common resources, such as a peripheral device. The simplest example of a resource scheduler is shown below:

```
monitor resource
var free: boolean

procedure request
when free: free:= false end

procedure release
begin free:= true end

begin free:= true end
```

The resource is free to begin with. A request operation delays the calling process until the resource is free and makes it available exclusively for that process. A release operation makes the resource free again.

A study of three model operating systems written in Concurrent Pascal shows that most monitors either serve as buffers or as schedulers [2].

In this discussion the details of the programming language are not important. We will just assume that a concurrent program consists of a fixed set of monitors M followed by a fixed set of processes P1, P2, ... , Pn. The monitors are initialized before the processes are executed.

3. Single-processor system

We will describe three computer architectures for a concurrent programming language. These computers will use very similar methods of store allocation, but will gradually increase the degree of multiprocessing from 1 to 10 and, finally, to 100.

A single-processor system that has already been implemented is described first. Its purpose is to introduce the storage allocation scheme and to characterize the performance of a single processor programmed in Concurrent Pascal.

On a single processor with a single store the code and variables of all monitors can be stored in a single segment M. The code and variables of each process can be stored in single segments P1, P2, ... , Pn (Fig. 2).

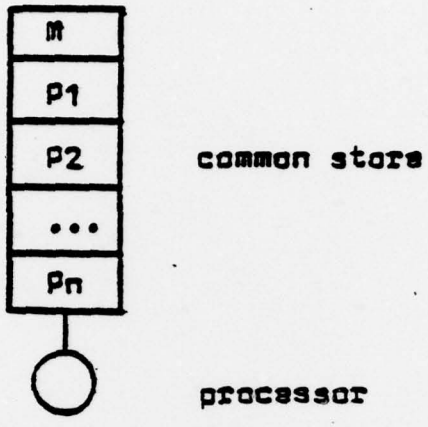


Fig. 2. Single processor system

The compiler ensures that processes only communicate by means of monitors. So each process only needs access to a subset of the whole store. This is important on a processor with a short word length (16 bits). It makes it possible to extend the store beyond the addressing limit by means of an address map that enables each process to see a virtual store consisting of the common segment M and its local segment Pi. This corresponds closely to the implementation of Concurrent Pascal on the PDP 11/45 computer [2].

On a single processor concurrent processes are simulated by means of clock interrupts. A process has exclusive access to the common segment as long as it performs a monitor operation without delaying itself. This is ensured by disabling clock interrupts temporarily. Processes that are ready to continue or are waiting for conditions to be satisfied are rescheduled periodically in round-robin order. This is the classical technique of processor multiplexing.

The Concurrent Pascal compiler generates virtual code for a stack machine. This virtual code is interpreted by a machine program of 1 K words on the PDP 11/45.

So far, three model operating systems have been written in Concurrent Pascal. The largest one is the Solo system which requires a store of 39 K words [2]. The monitors occupy a common segment of about 4 K words while the process segments vary from 200 words to 20 K words each.

One of our main concerns will be to avoid congestion of the common stores in a multiprocessor system. To illustrate the problem we will consider text processing as an example that involves a fair amount of data transmission between processes and a minimal amount of data processing. The processing of a stream of input data is common in real-time applications (although the data items may represent measurements rather than characters).

In the Solo system, the internal speed of text processing is about 1000 - 3000 char/sec for lexical analysis and line-oriented editing. In performing these tasks the system spends less than 10 per cent of its time on monitor operations. So the processes are certainly loosely connected.

4. A two-level multiprocessor

Figure 3 shows a proposed multiprocessor system with n microprocessors and n + 1 store modules. Each processor is dedicated to the execution of a single process. When a process delays itself its processor is also idle. A processor has a local store that holds the code and variables of a single process. The processors are connected to a single common store that holds the code and variables of all monitors.

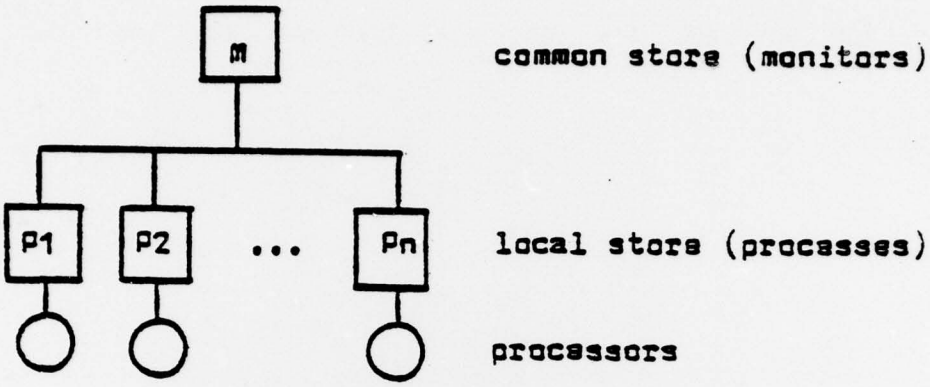


Fig. 3. A two-level multiprocessor

The virtual address space of a single processor consists of its local store and the common store. A processor can access its own store directly (but no other processor can). Access to the common store is controlled by a round-robin arbiter. Initially, we will assume that a processor has exclusive access to the common store for the duration of a monitor operation (or until the operation delays itself). This is achieved by request and release microinstructions on the common arbiter. From the point of view of a processor, its local store and the common store operate at the same speed (say, 3 μsec/word).

A peripheral device can be attached either to the local store of a single processor or to the common store. A processor remains idle during an input/output operation. This makes input/output appear to be an indivisible operation and eliminates the need for interrupts.

Fast response to external events is guaranteed by using dedicated processors that respond immediately to these events. As long as the common store is used primarily for internal communication (but is not a critical factor in the immediate response to real-time events) it is quite acceptable that it is monopolized by a single processor during a monitor operation. Later we will relax things a bit and permit operations on different monitors to take place simultaneously.

To make an evaluation of such a system we will assume that the processor and store technology is comparable to that of the LSI 11 microcomputers. If the virtual code generated by the compiler must be executed by an interpreter written in machine code then each processor will be 3 times slower than the PDP 11/45 [9]. But, if the virtual code is executed directly by microprograms, then it should be slightly faster than the interpreted code on the PDP 11/45. We will assume the latter and use the known performance figures for Concurrent Pascal statements [2] reduced somewhat to take the absence of processor multiplexing into account.

The send and receive operations on the buffer defined earlier will then take about (70 + 3c) μsec each, where c is the number of characters per message. So it will take 0.7 msec to transmit a block of 100 characters from one process to another. If the common store is used only for transmitting blocks of this size by means of monitors then it has a capacity of 135,000 char/sec.

To evaluate a case where the total traffic through the common store is high we assume that all the processors operate on characters at the highest possible rate. Consider therefore a machine with 10 processors and assume that each processor has a throughput of 3000 char/sec - a total throughput of 30,000 char/sec. This means that each processor uses the common store only 2 per cent of the time while all of them use it for only 20 per cent of the time. It is this strong localization of references to the local stores which makes the two-level system practical.

With a utilization factor of only 0.2 each processor will on the average immediately get access to the common store when it needs it [5]. Since the common store only comprises 9 per cent of the whole store, its low utilization does not matter.

In practice, it seems unlikely that an application will make it possible for 10 processors to operate continuously without idle periods. So the common store should certainly have sufficient speed for such a system.

In some applications processes might interact more often than in the previous example. In that case, it may be necessary to reduce the number of processes connected to a single common store somewhat and use a hierarchy of common stores as described later. However, if each process spends more than 10 per cent of its time in a common store, then we have a tightly coupled system which the machine was not built to handle.

Identical processes that operate in unison on common arrays is an important example of tight coupling. Another example is a pipeline that performs very fast operations on a stream of small data items. These special applications require a different kind of computer architecture which may be highly specialized if extreme speed is required [10]. This paper concentrates on computer architectures for a wide variety of applications in which different processes operate asynchronously at medium speeds. It may well be feasible to handle fast real-time applications by connecting a general-purpose multiprocessor to special-purpose processors for data reduction or synchronous computation.

5. The overhead of synchronization

So far we have ignored the problem of the periodic evaluation of a boolean expression B within a delayed monitor operation. When a process attempts to execute the synchronization statement

when B: S end

the process is in the middle of a monitor operation and has exclusive access to the common store. The code generated for the when statement will therefore be equivalent to the following program piece

```
while not B: release; request end
S
```

This code sequence evaluates the boolean expression. As long as it is false the processor releases the common store again and

waits for another turn. When another process has completed a monitor operation that makes the expression true, the statement S will be executed. The request and release operations refer to the arbiter that gives a processor exclusive access to the common store.

Typical synchronizing conditions, such as

not full length < max user in turn

will take 30-100 μsec each time they are reevaluated.

Previously, we considered the extreme case in which 10 processors were running continously without delay. We will now examine the other extreme case in which 9 processes are waiting for different conditions to be satisfied. When the last process executes a send operation on a buffer it becomes possible for one of the waiting processes to complete a receive operation. In that case, the send and receive operations can now both be delayed by the reevaluation of 8-9 conditions of 100 μsec each. This increases the response time of the interaction from 0.7 to 2.4 msec in the worst case. However, at the processing rate of 3000 char/sec, the exchange of a block of 100 characters only takes place every 33 msec. The only effect of the 2.4 msec is to slow the processes down by 7 per cent.

Since the arbiter interleaves the reevaluation of conditions with new monitor calls in round-robin order, the amount of reevaluation is automatically reduced when the traffic intensity increases.

Nevertheless, the reevaluation still has several undesirable consequences: (1) it makes the coupling of interacting processes unnecessarily tight at times; (2) it slows all other monitor operations down; and (3) it forces the programmer to use only the simplest possible conditions (rather than the most natural ones). It would therefore be desirable to limit the effects of reevaluation.

A radical (but expensive) solution to the problem of reevaluation is to use a common store that is 10 times faster than a local store and use local buffer registers to interleave references to single words in the common store. If we restrict the machine to 10 processors, the common store will always appear to be as fast as the local store of each processor. So the effect of reevaluation disappears.

Since we are now interleaving operations on different monitors simultaneously, we need a separate arbiter for each monitor. Each arbiter will be represented by a gate variable in the common store. The lock and unlock operations on a gate variable will be performed by microprograms. These operations are made indivisible by means of request and release operations on the single hardware arbiter:

```
lock(gate): request
            while gate = 0: release; request end
            gate:= 0
            release

unlock(gate): request; gate:= 1; release
```

If the common and local stores have the same access times the following scheme will limit the overhead of synchronization. Each monitor is now represented by two common variables, called the gate and the clock. The gate variable controls arbitration by means of lock and unlock operations as before. The clock is incremented cyclically by one every time an operation has changed the monitor variables. The clock need only be incremented at the beginning of each when statement and at the end of each monitor procedure.

Instead of reevaluating a complicated boolean expression B repeatedly a process can now simply look at the monitor clock until it changes its value. Only then is it necessary to evaluate the boolean expression again. This leads to the following code sequence for a when statement:

```
increment(clock)
while not B:
  unlock(gate)
  awaitchange(clock)
  lock(gate)
end
S
```

A delayed process stores the current value of the clock in a local register and compares it with the clock variable every 100 μsec. So the awaitchange instruction is microprogrammed as follows

```
register:= clock
while register = clock: idle(100) end
```

If we assume that load and store operations on the common store take place one at a time (thanks to the hardware arbiter), then it is not necessary for a delayed process to lock the gate variable just to look at the clock value. Consequently, the idling and reexamination of the clock variable only requires one cycle of the common store (or 3 μsec) every 100 usec.

As long as the state of a monitor is unchanged a delayed process can at most consume 3 per cent of the common store cycles. The reevaluation of a synchronizing condition B only takes place when another process changes the state of a monitor (and its clock). In the text processing example considered earlier an expression is evaluated in 100 μsec every 33 msec - an overhead of only 0.3 per cent. Within certain limits the overhead of synchronization is now influenced very little by the complexity of the expressions used. The existence of the gate and clock variables is hidden from the programmer.

The interleaving of common store references also makes it practical to connect a small number of shared peripherals to the common store (although we expect that most devices will be connected to the local store of a microprocessor).

6. A hierarchical multiprocessor

For real-time applications that require more than 10 processors we propose a block-structured programming language in which subsets of processes and monitors can be grouped into a hierarchy of subsystems.

A concurrent program now consists of nested subsystems (Fig. 4). Each subsystem in turn contains a set of monitors and processes. A process can use only those monitors that are within its own subsystem and within the subsystems that enclose it. In figure 4 the outer subsystem consists of the set of monitors M0. The first inner subsystem consists of the set of monitors M1 and the processes P1, P2, .. , Pm. These processes can use the local monitors M1 and the global monitors M0. Similarly, a process Qi in the other inner subsystem can use its local monitors M2 and the global monitors M0. But a process within one of the inner subsystems cannot use a monitor within the other inner subsystem.

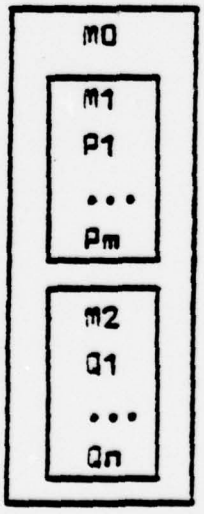


Fig. 4. A hierarchy of subsystems

We assume that each subsystem uses its own monitors an order of magnitude more frequently than it uses the global monitors.

Programs written in such a language can be executed by a multiprocessor with a hierarchy of common stores (Fig. 5).

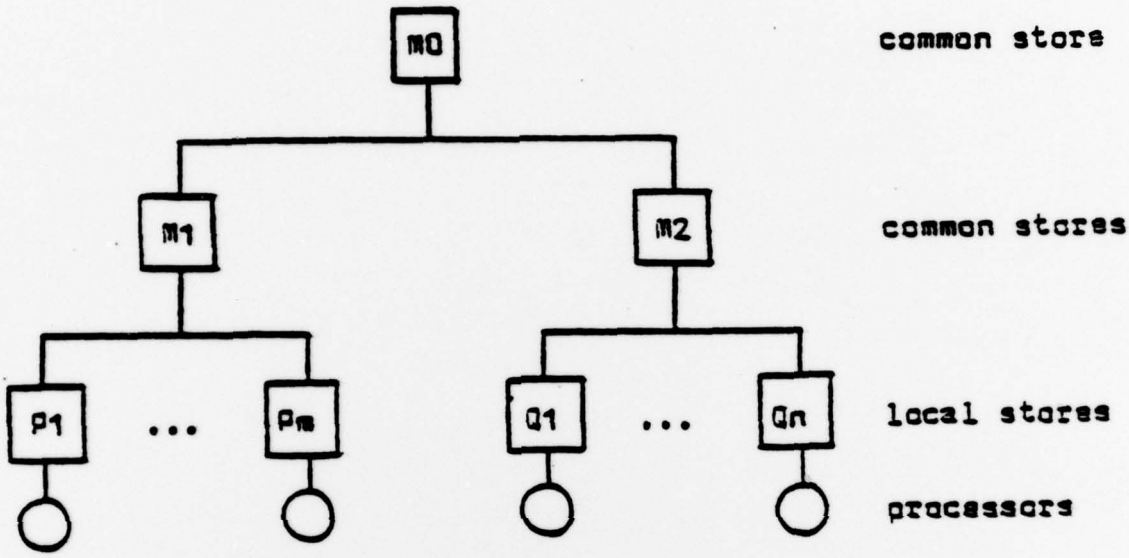


Fig. 5. A hierarchical multiprocessor

Although the horizontal lines can be interpreted as bus lines, Fig. 5 is <u>not</u> a diagram of the connections of hardware modules to bus lines. It is a diagram of the access rights of processors to store modules.

The virtual store of each processor consists of its local store and all the common stores that lie on a path from the processor to the root of the storage tree. For process P1 the common stores are M1 and M0. When P1 refers to M0 it has exclusive access to both M1 and M0. The hierarchical usage of arbiters prevents some deadlocks [5].

Although the multiprocessor in Fig. 5 seems to be tailored to the program sketched in Fig. 4, it should really be viewed as a general-purpose machine that can execute any concurrent program

with one or two subsystems which in turn are divided into no more than 10 processes each.

For a 16 bit processor it seems reasonable to have a three-level machine where the common stores contain 8 K words each and the local stores contain 16 K words each. Such a machine could include a total of 100 processors and 1.6 M words. A 32 bit processor system could have more processors and more store levels. It seems likely, however, that special-purpose machines are needed to utilize a much higher degree of concurrency efficiently.

7. Final remarks

The recent reduction of hardware costs for microprocessors will soon put great pressure on software designers to reduce their costs as well. The only way to do that is to write all software in abstract programming languages that hide the irrelevant details of computers. To make an abstract language efficient enough for real-time applications one must design a computer architecture that supports the language features directly.

Ten years ago this approach led to the development of stack machines for sequential programming languages. This paper suggests that a multiprocessor system with hierarchical storage will support a concurrent programming language with processes and monitors efficiently. Machines with stacks and tree-structured storage exploit the scope rules of the programming language to share storage efficiently among program modules.

The paper describes a reasonably simple way of limiting the reevaluation of synchronizing expressions within monitors. It also proposes a block-structured language concept (called a subsystem) which enables the programmer to partition the data structures of a concurrent program hierarchically among asynchronous processes.

It needs to be stressed again that this paper is only a proposal for a multiprocessor system that has not been built yet.

Acknowledgement

This work has been supported by the Office of Naval Research under contract NR049-415. It is a pleasure to acknowledge the helpful comments of Wolfgang Franzen, Charles Hayden, and Jørgen Staunstrup.

References

1. Hoare, C.A.R., Communicating sequential processes. To appear in Comm. ACM.

2. Brinch Hansen, P., The architecture of concurrent programs. Prentice-Hall, Englewood Cliffs, NJ, July 1977.

3. Swan, R.J., et al., Cm* - a modular multi-microprocessor. AFIPS Computer Conference 1977, 638-44.

4. Pattee, H.H. (ed.), Hierarchy theory - The challenge of complex systems. George Braziller, New York, NY, 1973.

5. Brinch Hansen, P., Operating system principles. Prentice-Hall, Englewood Cliffs, NJ, July 1973.

6. Hoare, C.A.R., Monitors: an operating system structuring concept. Comm. ACM 17, 10 (Oct. 1974), 549-57.

7. Hoare, C.A.R., Towards a theory of parallel programming. In Operating systems techniques, Academic Press, New York, NY, 1973.

8. Brinch Hansen, P., Structured multiprogramming. Comm. ACM 15, 7 (July 1972), 574-78.

9. Brinch Hansen, P., and Hayden, C. Microcomputer evaluation. Computer Science Department, University of Southern California, Los Angeles, California, Jan. 1978.

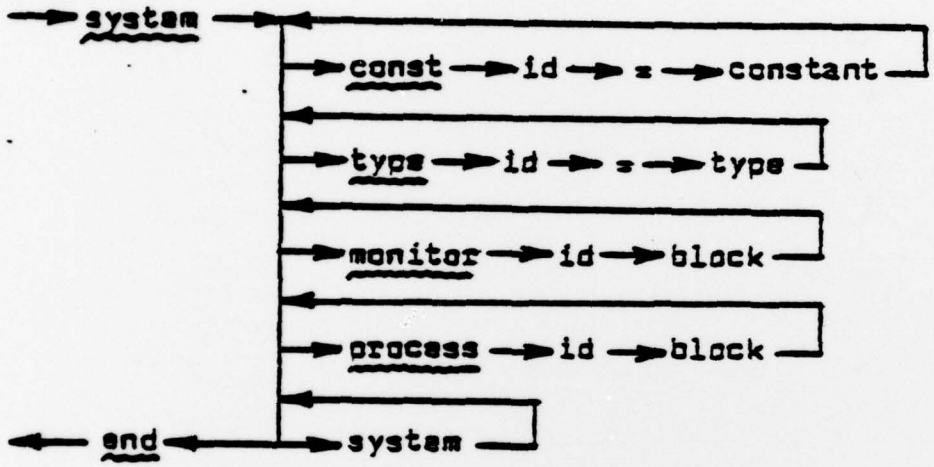
10. Stone, H.S. (ed.), Introduction to computer architecture. Science Research Associates, Chicago, ILL, 1975.

Appendix: Language syntax

This is an outline of the syntax of a concurrent programming language with nested subsystems containing monitors and processes.

program

system



block