DISTRIBUTED PROCESSES: A CONCURRENT PROGRAMMING CONCEPT

PER BRINCH HANSEN

(1978)

A language concept for concurrent processes without common variables is introduced. These processes communicate and synchronize by means of procedure calls and guarded regions. This concept is proposed for real-time applications controlled by microcomputer networks with distributed storage. The paper gives several examples of distributed processes and shows that they include procedures, coroutines, classes, monitors, processes, semaphores, buffers, path expressions, and input/output as special cases.

1 INTRODUCTION

This paper introduces *distributed processes*—a new language concept for concurrent programming. It is proposed for real-time applications controlled by microcomputer networks with distributed storage. The paper gives several examples of distributed processes and shows that they include procedures, coroutines, classes, monitors, processes, semaphores, buffers, path expressions and input/output as special cases.

Real-time applications push computer and programming technology to its limits (and sometimes beyond). A real-time system is expected to monitor simultaneous activities with critical timing constraints continuously and reliably. The consequences of system failure can be serious.

P. Brinch Hansen, Distributed processes: A concurrent programming concept, *Communications of the ACM 21*, 11 (November 1978), 934–941.

Real-time programs must achieve the ultimate in simplicity, reliability, and efficiency. Otherwise one can neither understand them, depend on them, nor expect them to keep pace with their environments. To make real-time programs manageable it is essential to write them in an abstract programming language that hides irrelevant machine detail and makes extensive compilation checks possible. To make real-time programs efficient at the same time will probably require the design of computer architectures tailored to abstract languages (or even to particular applications).

From a language designer's point of view, real-time programs have these characteristics:

1. A real-time program interacts with an environment in which many things happen simultaneously at high speeds.

2. A real-time program must respond to a variety of *nondeterministic requests* from its environment. The program cannot predict the order in which these requests will be made but must respond to them within certain time limits. Otherwise, input data may be lost or output data may lose their significance.

3. A real-time program controls a computer with a fixed configuration of processors and peripherals and performs (in most cases) a fixed number of concurrent tasks in its environment.

4. A real-time program never terminates but continues to serve its environment as long as the computer works. (The occasional need to stop a real-time program, say at the end of an experiment, can be handled by ad hoc mechanisms, such as turning the machine off or loading another program into it.)

What is needed then for real-time applications is the ability to specify a fixed number of concurrent tasks that can respond to nondeterministic requests. The programming languages *Concurrent Pascal* and *Modula* come close to satisfying the requirements for abstract concurrent programming (Brinch Hansen 1975, 1977; Wirth 1977). Both of them are based on the *monitor* concept (Brinch Hansen 1973; Hoare 1974). Modula, however, is primarily oriented towards multiprogramming on a single processor. And a straightforward implementation of Concurrent Pascal requires a single processor or a multiprocessor with a common store. In their present form, these languages are not ideal for a microcomputer network with distributed storage only.

It may well be possible to modify Concurrent Pascal to satisfy the constraints of distributed storage. The ideas proposed here are more attractive, however, because they unify the monitor and process concepts and result in more elegant programs. The new language concepts for real-time applications have the following properties:

1. A real-time program consists of a fixed number of concurrent processes that are started simultaneously and exist forever. Each process can access its *own variables* only. There are no common variables.

2. A process can call *common procedures* defined within other processes. These procedures are executed when the other processes are waiting for some conditions to become true. This is the only form of process communication.

3. Processes are synchronized by means of nondeterministic statements called *guarded regions* (Hoare 1972; Brinch Hansen 1978).

These processes can be used as program modules in a multiprocessor system with common or distributed storage. To satisfy the real-time constraints each processor will be dedicated to a single process. When a processor is waiting for some condition to become true then its processor is also waiting until an external procedure call makes the condition true. This does not represent a waste of resources but rather a temporary lack of useful work for that processor. Parameter passing between processes can be implemented either by copying within a common store or by input/output between separate stores.

The problems of designing verification rules and computer architectures for distributed processes are currently being studied and are not discussed. This paper also ignores the serious problems of performance evaluation and fault tolerance.

2 LANGUAGE CONCEPTS

A concurrent program consists of a fixed number of sequential processes that are executed simultaneously. A *process* defines its own variables, some common procedures, and an initial statement

process name
own variables
common procedures
initial statement

A process may only access its *own variables*. There are no common variables. But a process may call *common procedures* defined either within itself or within other processes. A procedure call from one process to another is called an *external request*.

A process performs two kinds of *operations* then: the *initial statement* and the *external requests* made by other processes. These operations are executed one at a time by *interleaving*. A process begins by executing its initial statement. This continues until the statement either terminates or waits for a condition to become true. Then another operation is started (as the result of an external request). When this operation in turn terminates or waits the process will either begin yet another operation (requested by another process) or it will resume an earlier operation (as the result of a condition becoming true). This interleaving of the initial statement and the external requests continues forever. If the initial statement terminates, the process continues to exist and will still accept external statements.

So the interleaving is controlled by the program (and *not* by clock signals at the machine level). A process switches from one operation to another only when an operation terminates or waits for a condition within a guarded region (introduced later).

A process continues to execute operations except when all its current operations are delayed within guarded regions or when it makes a request to another process. In the first case, the process is idle until another process calls it. In the second case, the process is idle until the other process has completed the operation requested by it. Apart from this nothing is assumed about the order in which a process performs its operations.

A process guarantees only that it will perform *some* operations as long as there are any unfinished operations that can proceed. But only the programmer can ensure that *every* operation is performed within a finite time.

A *procedure* defines its input and output parameters, some local variables perhaps, and a statement that is executed when it is called.

proc name(input param#output param)
local variables
statement

A process P can call a procedure R defined within another process Q as follows:

call $Q.R$(expressions, variables)

Before the operation R is performed the expression values of the call are assigned to the *input* parameters. When the operation is finished the values of the *output* parameters are assigned to the variables of the call. Parameter passing between processes can therefore be implemented either by copying within a common store or by input/output between processors that have no common store.

In this paper processes can call procedures within one another without any restrictions. In a complete programming language additional notation would be added to limit the access rights of individual processes. It may also be necessary to eliminate recursion to simplify verification and implementation. But these are issues that will not concern us here.

Nondeterminism will be controlled by two kinds of statements called *guarded commands* and *guarded regions*. A guarded region can delay an operation, but a guarded command cannot.

A guarded command (Dijkstra 1975) enables a process to make an arbitrary choice among several statements by inspecting the current state of its variables. If none of the alternatives are possible in the current state the guarded command cannot be executed and will either be skipped or cause a program exception.

The guarded commands have the following syntax and meaning:

if B_1: S_1 | B_2: S_2 | ... **end**

do B_1: S_1 | B_2: S_2 | ... **end**

If statement: If some of the conditions B_1, B_2, ..., are true then select one of the true conditions B_i and execute the statement S_i that follows it; otherwise, stop the program.

(If the language includes a mechanism whereby one process can detect the failure of another process, it is reasonable to let an exception in one process stop that process only. But, if recovery from programming errors is not possible then it is more consistent to stop the whole program. This paper does not address this important issue.)

Do statement: While some of the conditions are true, select one of them arbitrarily and execute the corresponding statement.

A guarded region (Hoare 1972; Brinch Hansen 1978) enables a process to wait until the state of its variables makes it possible to make an arbitrary choice among several statements. If none of the alternatives are possible in the current state the process postpones the execution of the guarded region.

The guarded regions have the following syntax and meaning:

when B_1: S_1 | B_2: S_2 | ... **end**

cycle B_1: S_1 | B_2: S_2 | ... **end**

When statement: Wait until one of the conditions is true and execute the corresponding statement.

Cycle statement: Endless repetition of a when statement.

If several conditions are true within a guarded command or region it is unpredictable which one of the corresponding statements the machine will select. This uncertainty reflects the nondeterministic nature of real-time applications.

The *data types* used are either integers, booleans, or characters, or they are finite sets, sequences, and arrays with at most n elements of some type T:

int bool char **set**$[n]T$ **seq**$[n]T$ **array**$[n]T$

The following statement enumerates all the elements in a data structure:

for x **in** y: S **end**

For statement: For each element x in the set or array y execute the statement S. A for statement can access and change the values of array elements but can only read the values of set elements.

Finally, it should be mentioned that the empty statement is denoted *skip* and the use of semicolons is optional.

3 PROCESS COMMUNICATION

The following presents several examples of the use of these language concepts in concurrent programming. We will first consider communication between processes by means of procedure calls.

Example: Semaphore

A general semaphore initialized to zero can be implemented as a process *sem* that defines *wait* and *signal* operations.

```
process sem
s: int
proc wait when s > 0: s := s − 1 end
proc signal; s := s + 1
s := 0
```

The initial statement assigns the value zero to the semaphore and terminates. The process, however, continues to exist and can now be called by other processes

call sem.wait **call** sem.signal

Example: Message buffer

A buffer process stores a sequence of characters transmitted between processes by means of *send* and *receive* operations.

```
process buffer
s: seq[n]char
proc send(c: char) when not s.full: s.put(c) end
proc rec(#v: char) when not s.empty: s.get(v) end
s := [ ]
```

The initial statement makes the buffer empty to begin with. The buffer operations are called as follows:

call buffer.send(x) **call** buffer.rec(y)

The semaphore and buffer processes are similar to *monitors* (Brinch Hansen 1973; Hoare 1974): They define the representation of a shared data structure and the meaningful operations on it. These operations take place one at a time. After initialization, a monitor is idle between external calls.

Example: Character stream

A process inputs punched cards from a card reader and outputs them as a sequence of characters through a buffer process. The process deletes *spaces* at the end of each card and terminates it by a *newline* character.

```
process stream
b: array[80]char; n, i: int
do true:
  call cardreader.input(b)
  if b = blankline: skip |
    b ≠ blankline: i := 1; n := 80;
      do b[n] = space: n := n − 1 end
      do i ≤ n: call buffer.send(b[i]); i := i + 1 end
  end
  call buffer.send(newline)
end
```

This use of a process is similar to the traditional *process* concept: the process executes an initial statement only. It calls common procedures within other processes, but does not define any within itself. Such a process does not contain guarded regions because other processes are unable to call it and make the conditions within it true.

The example also illustrates how *peripheral devices* can be controlled by distributed processes. A device (such as the card reader) is associated with a single process. Other processes can access the device only through common procedures. So a peripheral device is just another process.

While a process is waiting for input/output, no other operations can take place within it. This is a special case of a more general rule: When a process P calls a procedure R within another process Q then R is considered an indivisible operation within process P, and P will not execute any other operation until R is finished (see Section 2).

Notice, that there is no need for *interrupts* even in a real-time language. Fast response to external requests is achieved by dedicating a processor to each critical event in the environment and by making sure that these processors interact with a small number of neighboring processors only (to prevent them from being overloaded with too many requests at a time).

Exercise: Write a process that receives a sequence of characters from a buffer process and outputs them line by line to a printer. The process should output a *formfeed* after every 60 lines.

4 RESOURCE SCHEDULING

We will now look at a variety of scheduling problems solved by means of guarded regions. It should perhaps be mentioned that resource schedulers are by nature *bottlenecks*. It would therefore be wise in a real-time program to make sure that each resource either is used frequently by a small number of processes or very infrequently by a larger number of processes. In many applications it is possible to avoid resource scheduling altogether and dedicate a resource to a single process (as in the card reader and line printer examples).

Example: Resource scheduler

A set of user processes can obtain exclusive access to an abstract resource by calling request and release operations within a scheduling process.

```
process resource
free: bool
proc request when free: free := false end
proc release if not free: free := true end
free := true

call resource.request ... call resource.release
```

The use of the boolean *free* forces a strict alternation of request and release operations. The program stops if an attempt is made to release a resource that is already free.

In this example, the scheduler does not know the identity of individual user processes. This is ideal when it does not matter in which order the users are served. But, if a scheduler must enforce a particular scheduling policy (such as *shortest job next*) then it must know the identity of its users to be able to grant the resource to a specific user. The following example shows how this can be done.

Example: Shortest job next scheduler

A scheduler allocates a resource among n user processes in shortest job next order. A request enters the identity and service time of a user process in a queue and waits until that user is selected by the scheduler. A release makes the resource available again.

The scheduler waits until one of two situations arises:

1. A process enters or leaves the queue: The scheduler will scan the queue and select the next user (but will not grant the resource to it yet).

2. The resource is not being used and the next user has been selected: The scheduler will grant the resource to that user and remove it from the queue.

User processes identify themselves by unique indices 1, 2, ..., n. The constant *nil* denotes an undefined process index.

The scheduler uses the following variables:

queue	the indices of waiting processes
rank	the service times of waiting processes
user	the index of the current user (if any)
next	the index of the next user (if any)

```
process sjn
queue: set[n]int; rank: array[n]int
user, next, min: int

proc request(who, time: int)
begin queue.include(who); rank[who] := time
  next := nil; when user = who: next := nil end
end

proc release; user := nil

begin queue := [ ]; user := nil; next := nil
  cycle
    not queue.empty & (next = nil):
      min := maxinteger
      for i in queue:
        if rank[i] > min: skip |
          rank[i] ≤ min: next := i; min := rank[i]
        end
      end|
    (user = nil) & (next ≠ nil):
      user := next; queue.exclude(user)
  end
end
```

In a microprocessor network where each processor is dedicated to a single process it is an attractive possibility to let a process carry out computations *between* external calls of its procedures. The above scheduler takes advantage of this capability by selecting the next user while the resource is being used by the present user. It would be simpler (but less efficient) to delay the selection of the next user until the previous one has released the resource.

The scheduling of individual processes is handled completely by means of guarded regions without the use of synchronizing variables, such as semaphores or event queues.

The periodic reevaluation of a synchronizing condition, such as

$$\text{user} = \text{who}$$

might be a serious load on a *common* store shared by other processors. But it is quite acceptable when it only involves the *local* store of a single processor that has nothing else to do. This is a good example of the influence of hardware technology on abstract algorithms.

Exercise: Write a first-come, first-served scheduler.

Example: Readers and writers

Two kinds of processes, called readers and writers, share a single resource. The readers can use the resource simultaneously, but each writer must have exclusive access to it. The readers and writers behave as follows:

call resource.startread	**call** resource.startwrite
read	write
call resource.endread	**call** resource.endwrite

A variable s defines the current resource *state* as one of the following:

s = 0	1 writer uses the resource
s = 1	0 processes use the resource
s = 2	1 reader uses the resource
s = 3	2 readers use the resource
...	...

This leads to the following solution (Brinch Hansen 1978):

process resource
s: int
proc startread **when** s $\geq$ 1: s := s + 1 **end**
proc endread **if** s > 1: s := s − 1 **end**
proc startwrite **when** s = 1: s := 0 **end**
proc endwrite **if** s = 0: s := 1 **end**
s := 1

Exercise: Solve the same problem with the additional constraint that further reader requests should be delayed as long as some writers are either waiting for or are using the resource.

Example: Alarm clock

An alarm clock process enables user processes to wait for different time intervals. The alarm clock receives a signal from a timer process after each time unit. (The problems of representing a clock with a finite integer are ignored here.)

```
process alarm
time: int

proc wait(interval: int)
due: int
begin due := time + interval
  when time = due: skip end
end

proc tick; time := time + 1

time := 0
```

5 PROCESS ARRAYS

So far we have only used one instance of each process. The next example uses an array of n identical processes (Hoare 1978):

process name[n]

A standard function *this* defines the identity of an individual process within the array ($1 \leq \text{this} \leq n$).

Example: Dining philosophers

Five philosophers alternate between thinking and eating. When a philosopher gets hungry, he joins a round table and picks up two forks next to his plate and starts eaiting. There are, however, only five forks on the table. So a philosopher can eat only when none of his neighbors are eating. When a philosopher has finished eating he puts down his two forks and leaves the table again.

```
process philosopher[5]
do true: think
  call table.join(this); eat; call table.leave(this)
end

process table
eating: set[5]int
proc join(i: int)
when([i ⊖ 1, i ⊕ 1] & eating) = [ ]: eating.include(i) end
proc leave(i: int); eating.exclude(i)
eating := [ ]
```

This solution does not prevent two philosophers from starving a philosopher between them to death by eating alternately.

Exercise: Solve the same problem without starvation.

Example: Sorting array

A process array sorts m data items in order $O(m)$. The items are input through sort process 1 that stores the smallest item input so far and passes the rest to its successor sort process 2. The latter keeps the second smallest item and passes the rest to its successor sort process 3, and so on. When the m items have been input they will be stored in their natural order in sort processes 1, 2, ..., m. They can now be output in increasing order through sort process 1. After each output the processes receive the remaining items from their successors.

A user process behaves as follows:

```
A: array[m]int
for x in A: call sort[1].put(x) end
for x in A: call sort[1].get(x) end
```

The sorting array can sort n elements or less ($m \leq n$). A sorting process is in equilibrium when it holds one item only. When the equilibrium is disturbed by its predecessor, a process takes the following action:

1. If the process holds two items, it will keep the smallest one and pass the largest one to its successor.

2. If the process holds no items, but its successor does, then the process will fetch the smallest item from its successor.

A sorting process uses the following variables:

here	the items stored in this process ($0 \leq$ here.length ≤ 2)
rest	the number of items stored in its successors

A standard function *succ* defines the index of the successor process (succ = this + 1).

```
process sort[n]
here: seq[2]int; rest, temp: int
proc put(c: int) when here.length < 2: here.put(c) end
proc get(#v: int) when here.length = 1: here.get(v) end

begin here := [ ]; rest := 0
  cycle
    here.length = 2:
      if here[1] ≤ here[2]: temp := here[2]; here := [here[1]] |
        here[1] > here[2]: temp := here[1]; here := [here[2]]
      end
      call sort[succ].put(temp); rest := rest + 1 |
    (here.length = 0) & (rest > 0):
      call sort[succ].get(temp); rest := rest − 1
      here := [temp]
  end
end
```

A hardware implementation of such a sorting array could be used as a very efficient form of priority scheduling queue.

Exercise: Program a process array that contains $N = 2^n$ numbers to begin with and which will add them in time $O(\log_2 N)$.

Since a process can define a common procedure it obviously includes the *procedure* concept as a special case. Hoare (1978) shows that a process array also can simulate a *recursive* procedure with a fixed maximum depth of recursion.

Exercise: Write a process array that computes a Fibonacci number by recursion.

6 ABSTRACT DATA TYPES

A process combines a data structure and all the possible operations on it into a single program module. Since other processes can perform these operations only on the data structure, but do not have direct access to it, it is called an *abstract* data structure.

We have already seen that a process can function as a *monitor*—an abstract data type that is shared by several processes. The next example shows that a process also can simulate a *class*—an abstract data type that is used by a single process only.

Example: Vending machine

A vending machine accepts one coin at a time. When a button is pushed the machine returns an item with change provided there is at least one item left and the coins cover the cost of it; otherwise, all the coins are returned.

```
process vending_machine
items, paid, cash: int
proc insert(coin: int) paid := paid + coin
proc push(#change, goods: int)
if (items > 0) & (paid ≥ price)
      change := paid − price; cash := cash + price
      goods := 1; items := items − 1; paid := 0 |
   (items = 0) or (paid < price):
      change := paid; goods := 0; paid := 0
end
begin items := 50; paid := 0; cash := 0 end
```

7 COROUTINES

Distributed processes can also function as coroutines. In a coroutine relationship between two processes P and Q only one of them is running at a time. A resume operation transfers control from one process to the other. When a process is resumed it continues at the point where it has transferred control to another process.

```
process P
go: bool
proc resume; go := true

begin go := false
   ...
   call Q.resume
   when go: go := false end
   ...
end
```

Process Q is very similar.

8 PATH EXPRESSIONS

Path expressions define meaningful *sequences* of operations P, Q, ..., (Campbell 1974). A path expression can be implemented by a scheduling process

that defines the operations P, Q, ..., as procedures and uses a state variable s to enforce the sequence in which other processes may invoke these procedures.

Suppose, for example, that the operation P only can be followed by the operation Q as shown by the graph below:

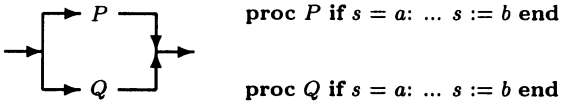
To implement this path expression one associates a distinct state a, b, and c with each arrow in the graph and programs the operations as follows:

proc P **if** $s = a$: ... $s := b$ **end**

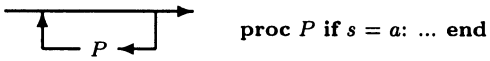
proc Q **if** $s = b$: ... $s := c$ **end**

If P is called in the state $s = a$ it will change the state to $s = b$ and make Q possible. Q, in turn, changes the state from b to c. An attempt to perform P or Q in a state where they are illegal will cause a program exception (or a delay if a *when* statement is used within the operation).

The next path expression specifies that either P or Q can be performed. This is enforced by means of two states a and b.



If an operation P can be performed zero or more times then the execution of P leaves the state $s = a$ unchanged as shown below.



The simple resource scheduler in Section 4 implements a composite path expression in which the sequence *request* ... *release* is repeated zero or more times.

The readers and writers problem illustrates the use of a state variable to permit some operations to take place *simultaneously* while other operations are temporarily *excluded* (in this case, simultaneous reading by several processes excludes writing). Each simultaneous operation P is surrounded by a pair of scheduling operations, *startP* and *endP*. The state variable counts the number of P operations in progress.

9 IMPLEMENTATION HINTS

The following outlines the general nature of an implementation of distributed processes but ignores the details which are currently being studied.

In a well-designed concurrent program one may assume that each process communicates with a small number of neighboring processes only. For if the interactions are not strongly localized one cannot expect to gain much from concurrency. (A few resource schedulers may be an exception to this rule.)

Each processor will contain a distributed process P and a small, fixed number of anonymous processes which are the *representatives* of those distributed processes that can call process P. Additional notation in the language should make it possible for a compiler to determine the number of processes which call a particular process.

Whenever a processor is idle it activates a local representative which then waits until it receives a request with input data from another processor. The representative now calls the local procedure requested with the available input. When the procedure terminates, its output data are returned to the other processor and the representative becomes passive again. The switching from one *quasiconcurrent* process to another within a processor takes place as described in Section 2.

Since processes are permanent and procedures are nonrecursive, a compiler can determine the maximum storage required by a distributed process and the local representatives of its environment. So the storage allocation is *static* within each processor.

The parameter passing between two processors requires a single *input* operation before a procedure is executed and a single *output* operation when it terminates.

The speed of process switching within a single processor will probably be crucial for its real-time response.

The technique of representing the environment of a processor by local processes synchronized with external processes seems conceptually attractive. Although these processes are anonymous in this proposal one could

design a language in which the store of a single process is shared by quasiconcurrent processes which communicate with nonlocal processes by input/output only.

10 FINAL REMARKS

It would certainly be feasible to adapt the processes and monitors of Concurrent Pascal to multiprocessor networks with distributed storage by restricting the parameter passing mechanism as proposed here. All the examples discussed here could then be programmed in that language—but not nearly as elegantly!

What then are the merits of distributed processes? Primarily, that they are a combination of *well-known* programming concepts (processes, procedures, and conditional critical regions) which *unify* the class, monitor, and process concepts. They include a surprising number of basic programming concepts as special cases:

procedures
coroutines
classes
monitors
processes
semaphores
buffers
path expressions
input/output

Since there is a common denominator for all these concepts, it may well be possible to develop common proof rules for them. The use of a single concept will certainly simplify the language implementation considerably.

The Concurrent Pascal machine distinguishes between 15 virtual instructions for classes, monitors, and processes. This number would be reduced by a factor of three for distributed processes. In addition, numerous special cases would disappear in the compiler.

It is also encouraging that distributed processes can be used to write elegant algorithms both for the more well-known concurrent problems and for some new ones that are nontrivial.

A recent proposal by Hoare (1978) has the same pleasant properties. Both proposals attack the problem of concurrency without shared variables and recognize the need for nondeterminacy within a single process.

Hoare's *communicating sequential processes* can be created and terminated dynamically. A single data transfer from one process to another is the communication mechanism. A process synchronizes itself with its environment by guarded input commands which are executed when a boolean expression is true *and* input is available from another process. The relationships between two communicating processes is symmetrical and requires both of them to name the other. The brief and nonredundant notation does not require declarations of communication channels but depends (conceptually) on dynamic type checking to recognize matching input and output commands in two processes.

In their present form communicating sequential processes seem well-suited to a theoretical investigation of concurrency and as a concise specification language that suppresses minor details, However, as Hoare points out, the language concepts and the notation would have to be modified to make them practical for program implementation.

The proposal for *distributed processes* is intended as a first step toward a practical language for networks. The proposal recognizes that the exchange of input and output in one operation is a frequent case, particularly for peripheral devices which return a result after each operation. The notation is redundant and enables a compiler to determine the number of processes and their storage requirements. The relationship between two communicating processes is asymmetrical and requires only that the caller of an operation name the process that performs it. This asymmetry is useful in hierarchical systems in which servants should be unaware of the identities of their masters.

Distributed processes derive much of their power from the ability to delay process interactions by means of boolean expressions which may involve both the global variables of a process *and* the input parameters from other processes (as illustrated by the *sjn* scheduler and the alarm clock). The price for this flexibility is the need for quasiconcurrent processes in the implementation. A more restricted form of Hoare's proposal might be able to implement process synchronization by the simpler method of polling a number of data channels until one of them transmits data.

But more work remains to be done on verification rules and network architectures for these new concepts. And then the ideas must be tested in *practice* before a final judgment can be made.

Acknowledgements

I am grateful to Nissim Francez, Wolfgang Franzen, Susan Gerhart, Charles Hayden, John Hennessy, Tony Hoare, David Lomet, David MacQueen, Johannes Madsen, David Musser, Michel Sintzoff, Jørgen Staunstrup and the referees for their constructive comments.

References

Brinch Hansen, P. 1973. *Operating System Principles.* Prentice Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. 1975. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering 1*, 2 (June), 199–207.

Brinch Hansen, P. 1977. *The Architecture of Concurrent Programs.* Prentice Hall, Englewood Cliffs, NJ.

Brinch Hansen, P., and Staunstrup, J. 1978. Specification and implementation of mutual exclusion. *IEEE Transactions on Software Engineering 4*, 4 (September), 365–370.

Campbell, R.H., and Habermann, A.N. 1974. The specification of process synchronization by path expressions. *Lecture Notes in Computer Science 16*, 89–102.

Dijkstra, E.W. 1975. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM 18*, 8 (August), 453–457.

Hoare, C.A.R. 1972. Towards a theory of parallel programming. In *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrott, Eds., Academic Press, New York.

Hoare, C.A.R. 1974. Monitors: An operating system structuring concept. *Communications of the ACM 17*, 10 (October), 549–557.

Hoare, C.A.R. 1978. Communicating sequential processes. *Communications of the ACM 21*, 8 (August), 666–677.

Wirth, N. 1977. Modula: A programming language for modular multiprogramming. *Software—Practice and Experience 7*, 1 (January), 3–35.