Monitors and Concurrent Pascal: A Personal History[*]

(1993)

This is a personal history of the early development of the monitor concept and its implementation in the programming language Concurrent Pascal. The paper explains how monitors evolved from the ideas of Dahl, Dijkstra, Hoare, and the author (1971–73). At Caltech the author and his students developed and implemented Concurrent Pascal and used it to write several model operating systems (1974–75). A portable implementation of Concurrent Pascal was widely distributed and used for system design (1976–90). The monitor paradigm was also disseminated in survey papers and text books. The author ends the story by expressing his own mixed feelings about monitors and Concurrent Pascal.

1 A Programming Revolution

In the 1970s new programming languages were developed to express asynchronous, concurrent processes. These languages support the now familiar paradigms for process communication known as *monitors*, *remote procedure calls*, and *synchronous communication.* The most influential early idea was the monitor concept and its implementation in the programming language *Concurrent Pascal.*

This is a personal history of how monitors and Concurrent Pascal were invented. I have tried to write the history of an *idea*—how it arose and spread through the scientific community. I have also described the struggles of the creative process, how you grope your way through obscurities and blind alleys until you find an elegant way of expressing an idea.

[*]P. Brinch Hansen, Monitors and Concurrent Pascal: A personal history. *2nd ACM Conference on the History of Programming Languages*, Cambridge, MA, April 1993. In *SIGPLAN Notices 28*, 3 (March 1993), 1–35. Copyright © 1993, Association for Computing Machinery, Inc.

The story of Concurrent Pascal frequently refers to my own work. However, I have let other researchers assess the merits and flaws of the language through quotations from the published literature. At the end of the paper I express my own reservations about the monitor concept and Concurrent Pascal. The appendix includes the personal comments of computer scientists who reviewed earlier drafts of this paper.

As someone who participated in these discoveries, I cannot claim to have written a complete and unbiased history of these events. In many cases my knowledge of related work is derived solely from the literature. I hope that historians will take care of these flaws by comparing my story with other sources.

From my perspective there are three distinct phases in the early history of monitors:

1971–73. Monitors evolved from the ideas of Ole-Johan Dahl, Edsger Dijkstra, Tony Hoare, and me. In 1973 Hoare and I independently published programming notations for monitors.

1974–75. Working with a few students and a professional programmer at Caltech, I developed and implemented the first programming language with monitors. My ambition was to do for operating systems what Pascal (and other programming languages) had done for compilers: to reduce the programming effort by an order of magnitude compared to assembly language. This was indeed achieved for small operating systems (but not for larger ones).

1976–90. A portable implementation of Concurrent Pascal was widely distributed and used for system design. The monitor paradigm was now disseminated throughout the computer science community in survey papers and textbooks on operating systems, concurrent programming, and programming languages.

Each phase will be described in a separate section of the paper.

After 1975 the monitor concept inspired other researchers to develop verification rules, monitor variants, and more programming languages (Andrews 1983). Originally I intended to include a broad review of these later developments, which I know only from the literature. However, after writing several earlier drafts of this paper, I realized that I can only provide meaningful historical remarks on ideas and events that I have first-hand knowledge about. The reviewers' comments confirmed this impression.

I will therefore limit the scope of this personal history to the discovery of the monitor concept and the development and use of the first monitor language Concurrent Pascal. Since I can only speak for myself, I have not imposed the same restriction on the reviewers' comments quoted in the appendix.

2 Monitors

On the road toward monitors several alternatives were considered. It may be easier to appreciate the piecemeal discovery if I briefly summarize the final idea. Monitors enable concurrent processes to share data and resources in an orderly manner. A monitor is a combination of shared variables and procedures. Processes must call these procedures to operate on the shared variables. The monitor procedures, which are executed one at a time, may delay the calling processes until resources or data become available.

Beginner's luck

I started out in industry as a systems programmer for the Danish computer manufacturer Regnecentralen in Copenhagen. In 1967 I became responsible for the development of the RC 4000 multiprogramming system.

In the 1960s most operating systems were huge, unreliable programs that were extremely difficult to understand and modify. The RC 4000 system was a radical departure from this state of affairs. It was not a complete operating system, but a small kernel upon which operating systems for different purposes could be built in an orderly manner. The kernel provided the basic mechanisms for creating a hierarchy of parallel processes that communicated by messages. The idea of designing a general kernel for operating system design was due to Jørn Jensen, Søren Lauesen, and me (Brinch Hansen 1969).

I consider myself lucky to have started in industry. The RC 4000 project convinced me that a fundamental understanding of operating systems would change computer programming radically. I was so certain of this that I decided to leave industry and become a researcher.

In November 1970 I became a Research Associate in the Department of Computer Science at Carnegie-Mellon University. My goal was to write the first comprehensive textbook on operating system principles (Brinch Hansen 1973b).

As soon as I started writing, it became clear that I needed an algorithmic language to express operating system functions concisely without unnecessary trivia. In an outline of the book I explained my choice of description language (Brinch Hansen 1971a):

> So far nearly all operating systems have been written partly or completely in machine language. This makes them unnecessarily difficult to understand, test and modify. I believe it is desirable and possible to write efficient operating systems almost entirely in a *high-level language.* This language must permit *hierarchal structuring* of data and program, extensive *error checking* at compile time, and production of *efficient machine code.*
>
> To support this belief, I have used the programming language *Pascal* throughout the text to define operating system concepts concisely by algorithms. Pascal combines the clarity needed for teaching with the efficiency required for design. It is easily understood by programmers familiar with Algol 60 or Fortran, but is a far more natural tool than these for the description of operating systems because of the presence of data structures of type record . . . and pointer.
>
> At the moment, Pascal is designed for sequential programming only, but I extend it with a suitable notation for multiprogramming and resource sharing.

Bold words indeed from a programmer who had never designed a programming language before, who did not have access to a Pascal compiler, and who had no way of knowing whether Pascal would ever be used for teaching! Niklaus Wirth (1971) had just published the first paper on Pascal, and there were, of course, no textbooks based on this new language.

A beautiful idea

The key problem in concurrent programming was to invent language concepts for asynchronous processes that share data in a common memory.

Dijkstra (1965) had argued that it is essential to treat operations on shared variables as *critical regions* that must be performed strictly one at a time in arbitrary order. He had also shown how to implement critical regions using semaphores. But he had not suggested a notation for this idea.

Hoare (1971b) proposed a notation that identifies a variable as a *resource* shared by concurrent processes. A good example is a ring buffer represented by an array with input/output indices and a message counter:

```
B: record inpointer, outpointer, count: integer;
     buffer: array 0..N−1 of T end;
{resource B; Producer//Consumer}
```

The buffer is shared by two concurrent processes which produce and consume messages, respectively. (In most examples, including this one, I have used the programming notation of the original papers.)

In his paper Hoare also introduced the elegant concept of a *conditional critical region* that is delayed until a resource satisfies a particular condition (defined by a Boolean expression).

The send operation on the ring buffer is a conditional critical region that is executed when the buffer is not full:

```
with B when count < N do
     begin buffer[inpointer] := next value;
           inpointer := (inpointer + 1) mod N;
           count := count + 1
     end
```

The receive operation is similar:

```
with B when count > 0 do
     begin this value := buffer[outpointer];
           outpointer := (outpointer + 1) mod N;
           count := count − 1
     end
```

A compiler must check that the resource is accessed only within critical regions. A computer must guarantee that these regions are executed one at a time without overlapping.

In retrospect, the limitations of conditional critical regions are perhaps obvious:

- The resource concept is unreliable.

The same variable may be treated as a scheduled resource in some contexts and as an ordinary variable in other contexts. This may enable one process to refer directly to a variable while another process is within a "critical" region on the same variable.

- The scheduling mechanism is too restrictive.

When a process is delayed by a Boolean expression without side effects, it cannot indicate the urgency of its request to other processes. This complicates the programming of priority scheduling.

- The context switching is inefficient.

It did not seem possible to implement conditional critical regions efficiently. The problem was to limit the repeated reevaluation of Boolean expressions until they became true.

- There is no precise idea of data abstraction.

The declaration of a resource and the operations associated with it are not combined into a single, syntactical form, but are distributed throughout the program text.

Attempts to remove these problems eventually led to the discovery of monitors.

Readers and writers

During the International Summer School in Marktoberdorf, Germany, July 1971, I removed the first two limitations of conditional critical regions by solving an interesting problem (Brinch Hansen 1972a).

Two kinds of processes, called *readers* and *writers*, share a single resource. Readers can use it simultaneously, but a writer can use the resource only when nobody else is using it. When a writer is ready to use the resource, it should be enabled to do so as soon as possible (Courtois 1971). This is, of course, a priority scheduling problem.

First, I solved a slightly simpler problem which permits several writers to use the resource simultaneously. A *shared variable* was now introduced by a single declaration:

```
var v: shared record readers, writers: integer end
```

This notation makes it clear that a shared variable may be accessed *only* within critical regions.

A reader waits until writers are neither using the resource nor waiting for it:

```
region v when writers = 0 do
  readers := readers + 1;
read;
region v do readers := readers − 1
```

A writer immediately announces itself and waits until no readers are using the resource:

```
region v do writers := writers + 1
  await readers = 0;
write;
region v do writers := writers − 1
```

The scheduling condition may appear at the beginning or at the end of a critical region. The latter permits *scheduling with side effects.* It was an obvious extension to permit a scheduling condition to appear anywhere within a critical region (Brinch Hansen 1972b).

Courtois (1972) and others had strong reservations about conditional critical regions and my solution to the readers and writers problem (Brinch Hansen 1973a).

A new paradigm

The idea of *monitors* evolved through discussions and communications between E.W. Dijkstra, C.A.R. Hoare, and me during the summer and fall of 1971. My own ideas were particularly influenced by our discussions at the International Summer School in Marktoberdorf, Germany, July 19–30, 1971. Hoare and I continued the exchange of ideas at the Symposium on Operating Systems Techniques in Belfast, August 30–September 3, 1971.

At Marktoberdorf, Dijkstra (1971) briefly outlined a paradigm of *secretaries* and *directors*:

> Instead of N sequential processes cooperating in critical sections via common variables, we take out the critical sections and combine them into a $N + 1^{st}$ process, called a “secretary”; the remaining N processes are called “directors.”
>
> A secretary presents itself primarily as a bunch of non-reentrant routines with a common state space.
>
> When a director calls a secretary ... the secretary may decide to keep him asleep, a decision that implies that she should

> wake him up in one of her later activities. As a result the identity of the calling program cannot remain anonymous as in the case of the normal subroutine. The secretaries must have variables of type "process identity." Whenever she is called the identity of the calling process is handed over in an implicit input parameter; when she signals a release—analogous to the return of the normal subroutine—she will supply the identity of the process to be woken up.

In Belfast I presented an outline of a course on operating system principles which included the following remarks (Brinch Hansen 1971a):

> The conceptual simplicity of simple and conditional critical regions is achieved by ignoring the sequence in which waiting processes enter these regions. This abstraction is unrealistic for heavily used resources. In such cases, the operating system must be able to identify competing processes and control the scheduling of resources among them. This can be done by means of a *monitor*—a set of shared procedures which can delay and activate individual processes and perform operations on shared data.

During a discussion on monitors I added (Discussion 1971):

> You can imagine the (monitor) calls as a queue of messages being served one at a time. The monitor will receive a message and try to carry out the request as defined by the procedure and its input parameters. If the request can immediately be granted the monitor will return parameters . . . and allow the calling process to continue. However, if the request cannot be granted, the monitor will prevent the calling process from continuing, and enter a reference to this transaction in a queue local to itself. This enables the monitor, at a later time when it is called by another process, to inspect the queue and decide which interaction should be completed now. From the point of view of a process a monitor call will look like a procedure call. The calling process will be delayed until the monitor consults its request. The monitor then has a set of scheduling queues which are completely local to it, and therefore protected against user processes. The latter can only access the shared variables maintained by the monitor

> through a set of well defined operations ... the monitor procedures.

At the Belfast Symposium, Hoare expressed his own reservations about conditional critical regions (Discussion 1971):

> As a result of discussions with Brinch Hansen and Dijkstra, I feel that this proposal is not suitable for operating system implementation.
>
> My proposed method encourages the programmer to ignore the question of which of several outstanding requests for a resource should be granted.
>
> The scheduling decision cannot always be expressed by means of a single Boolean expression without side effects. You sometimes need the power of a general procedural program with storage in order to make scheduling decisions. So it seems reasonable to take all these protected critical regions out, and put them together and call it a secretary or monitor.

In the 1960s the resident part of an operating system was often known as a *monitor*. The kernel of the RC 4000 multiprogramming system was called the monitor and was defined as a program that "can execute a sequence of instructions as an indivisible entity" (Brinch Hansen 1969).

At Belfast we discussed the disadvantages of the classical monitor written in assembly language (Discussion 1971):

> Brinch Hansen: The difficulty with the classical "monolithic" monitor is not the fact that while you are performing an operation of type A you cannot perform another operation of type A, but that if you implement them by a single critical section which inhibits further monitor calls then the fact that you are executing an operation A on one data set prevents all other operations on completely unrelated data sets. That is why I think the ability to have several monitors, each in charge of a single set of shared data, is quite important.
>
> Hoare: A monitor is a high-level language construction which has two properties which are not possessed by most monitors as actually implemented in machine code. Firstly, like all good

> programming ideas it can be called in at several levels: monitors can call other monitors declared in outer blocks. Secondly, the use of the high-level language feature enables you to associate with each monitor the particular variables and tables which are relevant for that monitor in controlling the relative progress of the processes under its care. The protection, which prevents processes from corrupting this information and prevents monitors from gaining access to information which has no relevance, is established by Algol-like scope rules.

These quotations show that Dijkstra, Hoare, and I had reached an informal understanding of monitors. But it was still no more than a verbal outline of the idea. The discovery of a queueing mechanism, a notation, and an implementation was left as an exercise for the reader.

Abandoned attempts

When a programming concept is understood informally, it would seem to be a trivial matter to invent a language notation for it. But in practice this is hard to do. The main problem is to replace an intuitive, vague idea with a precise, unambigious definition of its meaning and restrictions.

In the search for a suitable monitor notation many ideas were considered and rejected. I will describe two proposals that were abandoned. You may find them hard to understand. In retrospect, so do I!

At the Belfast Symposium, Hoare (1971a) distributed an unpublished draft of a monitor proposal which included a single-buffer characterized as follows:

status = −1	buffer empty (consumer waiting)
status = 0	buffer empty (consumer not waiting)
status = 1	buffer full (producer not waiting)
status = 2	buffer full (producer waiting)

The send operation is defined by a monitor entry named *p*:

```
p(prod) **entry**
**begin**
   **if** status ≤ 0 **then input** (prod)p(buffer);
   **if** status = −1 **then output** (cons)c(buffer);
   status := status + 1
**end**
```

This entry is not a procedure in the usual sense; *p* is the name of a communication between a producer and the buffer. The entry defines the protocol for this communication.

A producer outputs a message *e* to the buffer by executing the statement

output p(e)

The following takes place:

1. The producer is automatically delayed and its identity is assigned to a variable named *prod*.
2. If the buffer is empty, it immediately inputs the message from the producer and assigns it to a variable named *buffer* by executing the statement

   input (prod)p(buffer)

   The input automatically enables the producer to continue its execution.
3. If a consumer is waiting to input a message, the buffer immediately outputs the last message by executing the statement

   output (cons)c(buffer)

   The details of this statement will be explained shortly.
4. If the buffer is full it cannot input the message yet. In that case, the producer will remain delayed until a consumer empties the buffer as explained below.
5. Finally the buffer status is updated.

The receive operation is defined by a similar monitor entry named *c*:

```
c(cons) entry
begin
  if status $\geq$ 1 then output (cons)c(buffer);
  if status = 2 then input (prod)p(buffer);
  status := status $-$ 1
end
```

A consumer inputs a message and assigns it to a variable x by executing the statement

input c(x)

This has the following effect:

1. The consumer is automatically delayed and its identity is assigned to a variable named *cons*.
2. If the buffer is full, it immediately outputs the last message to the consumer by executing the statement

   output (cons)c(buffer)

3. If a producer is waiting to output a message to the buffer, the buffer now accepts that message by executing the statement

   input (prod)p(buffer)

4. If the buffer is empty, it cannot output a message yet. In that case, the consumer will remain delayed until a producer fills the buffer as explained earlier.
5. Finally the buffer status is updated.

The proposal offers an efficient mechanism for process scheduling. The basic idea is that one monitor entry can complete a communication that was postponed by another monitor entry. This is the programming style one naturally adopts in a monolithic monitor written in assembly language.

The description of parameter transfers as unbuffered input/output later became the basis for the concept of *communicating sequential processes* (Hoare 1978).

This early monitor proposal did not combine monitor entries and shared variables into a modular unit and did not specify parameter types. In an attempt to remedy these problems, I sent Hoare an unpublished draft of "a monitor concept which closely mirrors the way in which the RC 4000 monitor was programmed" (Brinch Hansen 1971c). Algorithm 1 illustrates the use of this notation to implement a single-buffer.

```
monitor
var send2: ref send; receive2: ref receive;
  ready: Boolean;

entry send(const x: message)
call send1;
begin
  if ready then
    complete send1, receive2 do
    begin y := x; ready := false end
  else
    begin send2 := send1; ready := true end
end

entry receive(var y: message)
call receive1;
begin
  if ready then
    complete receive1, send2 do
    begin y := x; ready := false end
  else
    begin receive2 := receive1; ready := true end
end

begin ready := false end
```

Algorithm 1 An abandoned proposal.

A monitor is now a module that combines shared variables, procedures and an initial statement. The latter must be executed before the monitor can be called.

When the producer calls the *send* procedure, the following happens:

1. A reference to the call is stored in a local variable named *send1*. This is called a send reference.

2. If the consumer has called the *receive* procedure and is ready to receive a message, the monitor completes the send and receive calls simultaneously by assigning the value parameter x in the send call to the variable parameter y in the receive call. The completion statement extends the

scope of the send entry with the parameters of the corresponding receive entry. It also has the side effect of resuming the two processes associated with the procedure calls.

3. If the consumer is not ready, the monitor stores the identity of the send call in a global variable named *send2* and indicates that the producer is ready to communicate.

The *receive* procedure is similar.

The use of *call references* enables a compiler to check parameter declarations in completion statements.

The most serious flaw of both proposals is the unreliable nature of process scheduling. As Hoare put it: "It would be a grave error for a monitor to specify an interaction with a process which was not waiting for that interaction to take place." I concluded that it is generally "impossible ... to check the validity of process references."

In a collection of papers by Hoare (1989), C.B. Jones introduces Hoare's 1974 paper on monitor and writes: "The first draft of this paper was distributed to the participants of the 1971 Belfast Symposium."

However, there is very little resemblance between these two papers. The reason is quite simple. In 1971 we had some understanding of abstract data types. But a key ingredient of monitors was still missing: a secure, efficient method of context switching. We now turn to this problem.

The waiting game

On February 16, 1972, I presented a completely different solution to the problem of process scheduling at the California Institute of Technology (Brinch Hansen 1972b).

I will illustrate the idea by an exercise from Brinch Hansen (1973b). Processes $P_1, P_2, \ldots, P_n$ share a single resource. A process requests exclusive access to the resource before using it and releases it afterwards. If the resource is free, a process may use it immediately; otherwise the process waits until another process releases the resource. If several processes are waiting for the resource, it is granted to the waiting process P_i with the lowest index i.

Algorithm 2 shows a priority scheduler for this problem. The resource is represented by a *shared record r*. The key idea is to associate *scheduling queues* with the shared variable. The queues are declared as variables of

```
var r: shared record
               free: Boolean;
               waiting: array [1..n] of Boolean;
               grant: array [1..n] of event r
             end

procedure request(i: 1..n);
region r do
begin
  if free then free := false
  else
    begin
      waiting[i] := true;
      await(grant[i]);
      waiting[i] := false
    end
end

procedure release;
var i, m: 1..n;
region r do
begin
  i := 1; m := n;
  while i < m do
    if waiting[i] then m := i
    else i := i + 1;
  if waiting[i] then cause(grant[i])
  else free := true
end
```

Algorithm 2 Context switching queues.

type *event r*. The resource scheduler can delay processes in these queues and resume them later by means of two standard procedures named *await* and *cause*.

If a process P_i calls the request procedure when the resource is not free, the Boolean *waiting[i]* is set to true and the process is entered in the event queue *grant[i]*. The await operation makes the process leave its critical region temporarily.

When a process P_j calls the *release* procedure while other processes are waiting, the most urgent process P_i is selected and enabled to resume as soon as P_j leaves its own region. At that moment P_i reenters its previous region and continues execution after the await statement.

Instead of letting one critical region complete the execution of another region, we simply switch back to the context of the previous region. Consequently, a scheduling decision can be viewed merely as a delay during the execution of a critical region.

This queueing mechanism enables the programmer to ignore the identity of a process and think of it only as "the calling process" or "the process waiting in this queue." There is no need for variables of type process reference.

The only possible operations on a queue are *cause* and *await*, performed within critical regions. The problem of dangling process references is solved by making the queues empty to begin with and preventing assignments to them.

My proposal included a feature that was never used. Suppose several processes are waiting in the same queue until a Boolean expression B is true. In that case, a *cause* operation on the queue enables *all* of them to resume their critical regions one at a time. Mutual exclusion is still maintained, and processes waiting to resume critical regions have priority over processes that are waiting to enter the beginning of critical regions. In this situation, a resumed process may find that another process has made the scheduling condition B false again. Consequently, processes must use waiting loops of the form

```
while not B do await(q)
```

My 1972 paper, which introduced scheduling queues, was an invited paper written under great time pressure. When someone later mentioned that multiple resumption might be inconvenient, I looked at the paper again and saw that it presented one example only of the use of scheduling queues. And that example used a separate queue for each process! The programming examples in my operating systems book (Brinch Hansen 1973b) did the same. In Concurrent Pascal I turned this programming style into a programming language rule (Brinch Hansen 1974d).

In spite of the unintended generality, my 1972 process queues were *not* the same as the classical event queues of the 1960s, which caused the programmer to lose control over scheduling. The crucial difference was that the new queues were associated with a shared variable, so that all scheduling operations were mutually exclusive operations. The programmer could control the scheduling of processes to any degree desired by associating each queue with a *group* of processes or an *individual* process.

The idea of associating scheduling queues with a shared variable to enable processes to resume critical regions was the basis of all subsequent monitor proposals. Context switching queues have been called *events* (Brinch Hansen 1972b), *queues* (Brinch Hansen 1973b), and *conditions* (Hoare 1973a). Some are single-process queues; others are multiprocess queues. The details vary, but they all combine process scheduling with context switching and mutual exclusion.

We now had all the pieces of the monitor puzzle. And I had adopted a programming style which combined shared variables, queues, critical regions, and procedures in a manner that closely resembled monitors (Algorithm 2).

A moment of truth

In the spring of 1972 I read two papers by Dahl (1972b) and Hoare (1972b) on the *class* concept of the programming language *Simula 67*. Although Simula is not a concurrent programming language, it inspired me in the following way: So far I had thought of a monitor as a program module that defines all operations on a *single* instance of a data structure. From Simula I learned to regard a program module as the definition of a *class* of data structures accessed by the same procedures.

This was a moment of truth for me. Within a few days I wrote a chapter on resource protection for my operating systems book. I proposed to represent monitors by *shared classes* and pointed out that resource protection and type checking are part of the same problem: to verify automatically that all operations on data structures maintain certain properties (called *invariants*).

My book includes the buffer monitor defined by Algorithm 3. The shared class defines a data structure of type B, two procedures which can operate on the data structure, and a statement that defines its initial state.

```
shared class B =
   buffer: array 0..max−1 of T;
   p, c: 0..max−1;
   full: 0..max;

   procedure send(m: T);
   begin
      await full < max;
      buffer[p] := m;
      p := (p + 1) mod max;
      full := full + 1;
   end

   procedure receive(var m: T);
   begin
      await full > 0;
      m := buffer[c];
      c := (c + 1) mod max;
      full := full − 1;
   end

begin p := 0; c := 0; full := 0 end
```

Algorithm 3 A monitor with conditional waiting.

The class notation permits multiple instances of the same monitor type. A buffer variable *b* is declared as follows:

var b: B

Upon entry to the block in which the buffer variable is declared, storage is allocated for its data components, and the buffer is initialized by executing the statement at the end of the class definition.

Send and *receive* operations on the buffer *b* are denoted

b.send(x) b.receive(y)

A shared class is a notation that explicitly restricts the operations on a data type and enables a compiler to check that these restrictions are obeyed.

It also indicates that all operations on a particular instance must be executed as critical regions.

In May 1972 I submitted the manuscript of my book to Prentice-Hall and sent copies to Dijkstra and Hoare. On November 3, 1972, I gave a seminar on shared classes at the University of California at Santa Barbara.

In July 1973 *Operating System Principles* was published with my monitor proposal based on Simula classes (Brinch Hansen 1973b). My decision to use conditional waiting in this proposal was a matter of taste. I might just as well have used queues, which I had introduced in another chapter.

I also included the monitor notation in the first draft of a survey paper on concurrent programming (Brinch Hansen 1973d). A referee, who felt that it was inappropriate to include a recent idea in a survey paper, suggested that I remove it, which I did.

I discussed monitors with queues in the first report on Concurrent Pascal, April 1974, and at the IFIP Congress in Stockholm, August 1974 (Brinch Hansen 1974a, 1974c).

Parallel discovery

Two influential papers concluded the early development of monitors. In the first paper Hoare (1973b) used a monitor in the design of a paging system. He begins the paper by acknowledging that "The notations used ... are based on those of Pascal ... and Simula 67." In the second paper Hoare (1974a) illustrated the monitor concept by several examples, including a ring buffer (Algorithm 4). Communicating processes are delayed and resumed by means of *wait* and *signal* operations on first-in, first-out queues called *condition* variables.

In an unpublished draft of his condition proposal Hoare (1973a) correctly pointed out that

> The synchronization primitives proposed here are very similar to Brinch Hansen's "await" and "cause", but they involve less retesting inside waiting operations, and may be slightly more efficient to implement.

According to Hoare (1989), his first monitor paper was submitted in October 1972; his second paper was submitted in February 1973 and the material presented at IRIA, Paris, France, on May 11, 1973. I received them shortly before they were published in August 1973 and October 1974, respectively.

```
bounded buffer: **monitor**
  begin buffer: array 0..N−1 of portion;
    lastpointer: 0..N−1;
    count: 0..N;
    nonempty, nonfull: condition;
    procedure append(x: portion);
      begin if count = N then nonfull.wait;
        note $0 \leq$ count $<$ N;
        buffer[lastpointer] := x;
        lastpointer := lastpointer $\oplus$ 1;
        count := count + 1;
        nonempty.signal
      end append;
    procedure remove(result x: portion);
      begin if count = 0 then nonempty.wait;
        note $0 <$ count $\leq$ N;
        x := buffer[lastpoint $\ominus$ count];
        count := count − 1;
        nonfull.signal
      end remove;
    count := 0; lastpointer := 0
  end bounded buffer;
```

Algorithm 4 A monitor with queues.

While writing this history I discovered a working paper by McKeag (1973) submitted to an ACM meeting in Savannah, Georgia, April 9–12, 1973. This early paper includes a single example of Hoare's monitor notation.

Milestones

The classical monitor of the 1960s was not a precisely defined programming concept based on rules enforced by a compiler. It was just a vague term for the resident part of an operating system, which was programmed in assembly language. The monitor concept that emerged in the 1970s should not be regarded as a refinement of an operating systems technique. It was a new programming language concept for concurrent programs running on shared-memory computers. Operating systems were just a challenging application area for this synchronization concept.

This brings us to the end of the phase where the monitor concept was discovered. The milestones were:

1971	Conditional critical regions
	Scheduling with side effects
	Monitor idea
1972	Context switching queues
	Class concept papers
	Monitor notation
1973	*Operating System Principles*
	Monitor papers

The next task was to develop a programming language with monitors.

3 Concurrent Pascal

Concurrent Pascal extended the sequential programming language Pascal with concurrent processes, monitors, and classes. The polished presentations of the language in professional journals and text books fail to show the long arduous road we had to travel to understand what undergraduates now take for granted.

A matter of philosophy

In designing Concurrent Pascal I followed a consistent set of principles for programming languages. These principles carried structured programming into the new realm of modular, concurrent programming. Let me summarize these principles and show *when* and *how* I first expressed them in writing.

- Concurrent programs can be written exclusively in high-level languages.

In the fall of 1971 I expressed this belief, which seems commonplace today, but was novel at the time (see the earlier quotation in "Beginner's luck"). Later I will explain why I did not consider Burroughs Algol and PL/I as high-level programming languages for operating system design.

In Brinch Hansen (1974c) I repeated the same idea:

> I am convinced that in most cases operating system designers do not need to control low-level machine features (such as registers, addresses, and interrupts) directly, but can leave these

> problems to a compiler and its run-time environment. A consistent use of abstract programming concepts in operating system design should enable a compiler to check the access rights of concurrent processes and make enforcement of resource protection rules at run time largely unnecessary.

Hoare (1971b) and Brinch Hansen (1971b) introduced a fundamental requirement of any concurrent programming language:

- Time-dependent programming errors must be detected during compilation.

In the spring of 1972 I explained this requirement as follows (Brinch Hansen 1973b):

> The main dificulty of multiprogramming is that concurrent activities can interact in a time-dependent manner which makes it practically impossible to locate programming errors by systematic testing. Perhaps, more than anything else, this explains the difficulty of making operating systems reliable.
>
> *If we wish to succeed in designing large, reliable multiprogramming systems, we must use programming tools which are so well-structured that most time-dependent errors can be caught at compile time.* It seems hopeless to try to solve this problem at the machine level of programming, nor can we expect to improve the situation by means of so-called "implementation languages," which retain the traditional "right" of systems programmers to manipulate addresses freely.

In 1976 I put it this way (Brinch Hansen 1977b):

> One of the primary goals of Concurrent Pascal is to push the role of *compilation checks* to the limit and reduce the use of *execution checks* as much as possible. This is not done just to make compiled programs more efficient by reducing the overhead of execution checks. In program engineering, compilation and execution checks play the same role as preventive maintenance and flight recorders do in aviation. The latter only tell you why a system crashed; the former prevents it. This distinction seems

essential to me in the design of real-time systems that will control vital functions in society. Such systems must be highly reliable *before* they are put into operation.

Time-dependent errors occur when processes refer to the same variables without proper synchronization. The key to preventing these *race conditions* turned out to be the requirement that

- A concurrent programming language should support a programming discipline that combines data and procedures into modules.

I realized this even *before* discovering a monitor notation. The following quotation refers to my earlier proposal of associating shared variables with critical regions and scheduling queues (Brinch Hansen 1972b):

> The basic idea is to associate data shared by concurrent processes explicitly with operations defined on them. This clarifies the meaning of programs and permits a large class of time-dependent errors to be caught at compile-time.

In the spring of 1972 I described my own monitor notation as a natural extension of the module concept of Simula 67 (Brinch Hansen 1973b):

> In Simula 67, the definition of a structured data type and the meaningful operations on it form a single syntactical unit called a *class*.
>
> An obvious idea is to represent critical regions by the concept *shared class*, implying that the operations ... on a given variable v of type T exclude one another in time.
>
> My main purpose here is to show a notation which explicitly restricts operations on data and enables a compiler to check that these restrictions are obeyed. Although such restrictions are not enforced by Simula 67, this would seem to be essential for effective protection.

Concurrent Pascal was the first realization of *modular, concurrent programming*. During the 1970s researchers also introduced modularity in sequential programming languages. However, these languages were completed and implemented *after* Concurrent Pascal (Popek 1977; Liskov 1981; Shaw 1981).

In the spring of 1975, after implementing Concurrent Pascal and writing the first operating system in the language, I wrote the following (Brinch Hansen 1975c):

> The combination of a data structure and the operations used to access it is called an *abstract data type.* It is abstract because the rest of the system only needs to know what operations one can perform on it but can ignore the details of how they are carried out. A Concurrent Pascal program is constructed from three kinds of abstract data types: processes, monitors, and classes.

Race conditions are prevented by a simple scope rule that permits a process, monitor, or class to access its own variables only. In a suitably restricted language this rule can easily be checked by a compiler. However, in a language with pointers and address arithmetic, no such guarantee can be offered.

The principles discussed so far were largely derived from my perception of concurrent programming in 1972. Intuitively I also followed a more general principle of language design, which I only formulated four years later:

- A programming language should be abstract and secure.

In the spring of 1976 I explained this requirement as follows (Brinch Hansen 1977b):

> The main contribution of a good programming language to simplicity is to provide an abstract *readable notation* that makes the parts and structures of programs obvious to a reader. An abstract programming language *suppresses machine detail* (such as addresses, registers, bit patterns, interrupts, and sometimes even the number of processors available). Instead the language relies on *abstract concepts* (such as variables, data types, synchronizing operations, and concurrent processes). As a result, program texts written in abstract languages are often an order of magnitude shorter than those written in machine language. This *textual reduction* simplifies program engineering considerably.

> We shall also follow the crucial principle of language design suggested by Hoare: *The behavior of a program written in an abstract language should always be explainable in terms of the concepts of that language and should never require insight into the details of compilers and computers.* Otherwise, an abstract notation has no significant value in reducing complexity.

A programming language that satisfies this requirement is said to be *secure* (Hoare 1974b).

A programming language that permits unrestricted use of assembly language features, such as jumps, typeless machine words, and addresses is insecure. A program written in such a language may have unpredictable effects that force the programmer to go beyond the abstract concepts, which the programming language pretends to support. In order to locate obscure programming errors, the programmer may now have to consider machine-dependent details, which vary from one computer to another (or even from one execution to another on the same computer).

The Burroughs B6700 and Multics operating systems were written in programming languages that permit unrestricted address manipulation (extended Algol 60 and PL/I). These insecure programming languages and operating systems had no influence on Concurrent Pascal and the model operating systems written in the language. The Unix system, written in the insecure language C, had not yet been described when Concurrent Pascal was being developed.

The controversy over whether a programming language should give you unrestricted access to hardware features or impose restrictions that simplify programs and facilitate error detection has continued to this day.

Facing complexity

On July 1, 1972, I became Associate Professor of Computer Science at the California Institute of Technology. During my first academic year I prepared three new courses and introduced Pascal on campus. These tasks kept me busy for a while.

I also started thinking about designing a programming language with concurrent processes and monitors. To reduce the effort, I decided to include these concepts in an existing sequential language. Pascal was an obvious choice for me, since I had used the language in my operating systems book. I liked Pascal because of its similarity to Algol 60, which I had used extensively at Regnecentralen. I named the new language *Concurrent Pascal* and did not consider any other base language. Apart from that, nothing else was obvious.

With a notation for monitors now in hand, you would think it would be easy to include it in Pascal. I had no idea of how to do this. I remember sitting in my garden in Pasadena, day after day, staring at a blank piece of paper and feeling like a complete failure.

Let me just mention some of the complicated issues I faced for the first time:

How can a programming language support

- The different scope rules of Pascal blocks and Simula classes?
- Hierarchical composition of processes and monitors?
- Multiple instances of the same process or monitor type?
- Dynamic activation and termination of processes and monitors?
- Elementary input/output from arbitrary peripherals?

How can a compiler check that

- Processes communicate by monitor procedures only?
- Monitors do not deadlock by calling themselves recursively (either directly or indirectly)?

How can a minicomputer with inadequate facilities for dynamic memory allocation

- Execute concurrent programs efficiently?

It took me almost two years to find reasonable solutions to most of these problems and make compromises which enabled me to ignore the most thorny issues.

A new language

In September 1973 and April 1974 I distributed the first descriptions of Concurrent Pascal. A final paper and a language report were both published in June 1975 (Brinch Hansen 1973c, 1974a, 1974d, 1975a).

I now understood what I was doing. One day the Caltech president, Harold Brown, came to my office and asked me to explain my research. After listening for half an hour, he said, "That sounds easy." I agreed because that was how I felt at the time.

A Concurrent Pascal program defines a fixed number of concurrent processes which communicate by monitors only. One of the first programs I wrote in Concurrent Pascal implements a pipeline that reads and prints an endless sequence of punched cards. Figure 1 shows the hierarchical structure of the pipeline. It consists of three processes connected by two line buffer monitors. An arrow from a process to a monitor indicates that the process can call that monitor. I named this kind of representation an *access graph.* It became our main tool for "programming in the large."

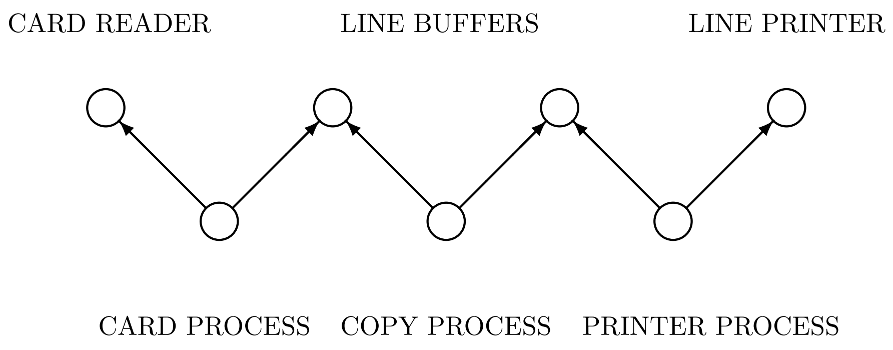

Figure 1 An access graph.

I will use the pipeline in Fig. 1 to illustrate the *syntax* and *semantics* of Concurrent Pascal.

Both *line buffers* in this pipeline are defined by the same *monitor type* (Algorithm 5). Each buffer can hold a single line at a time. A Boolean variable defines whether or not a buffer is full. Two variables of type *queue* are used to delay and continue the sender and receiver, respectively.

The pipeline program uses two line buffers, which are declared and ini-

```
type linebuffer =
monitor
var contents: line; full: Boolean;
    sender, receiver: queue;

procedure entry receive(var text: line);
begin
  if not full then delay(receiver);
  text := contents; full := false;
  continue(sender)
end;

procedure entry send(text: line);
begin
  if full then delay(sender);
  contents := text; full := true;
  continue(receiver)
end;

begin full := false end
```

Algorithm 5 A monitor type.

tialized as follows:

```
var inbuffer, outbuffer: linebuffer;

init inbuffer, outbuffer
```

For each buffer, the *init* statement allocates memory space for fresh instances of the shared variables declared at the beginning of the monitor type. The initialization also causes the statement at the end of the monitor to be executed, which makes a buffer empty to begin with. Each buffer is now ready to be shared by a sender and a receiver as shown in Fig. 1.

The Concurrent Pascal compiler checks that processes only access a line buffer by calling the monitor procedures *send* and *receive*. This restriction is guaranteed by a scope rule that makes the shared variables inaccessible from outside the monitor.

A sender outputs a line of text through a particular buffer by calling the *send* procedure as follows:

```
var text: line;

outbuffer.send(text)
```

If the buffer is full, the send procedure *delays* the calling process in the sender queue. The delay lasts until another process calls the *receive* procedure, which performs a *continue* operation on the sender queue. In any case, the sender cannot complete the *if* statement until the buffer is empty. At that point, the sender puts a message in the queue, performs a continue operation on the receiver queue and returns from the send procedure.

The *receive* procedure is similar.

While a process executes a monitor procedure, it has exclusive access to the shared variables. If another process attempts to call the same monitor while a process has exclusive access to that monitor, the latter call will automatically be delayed until the former process has released its exclusive access.

A process releases its exclusive access to a monitor in one of three ways:

1. By reaching the end of a monitor procedure.
2. By delaying itself temporarily in a queue declared within the monitor. The process regains its exclusive access when another process performs a continue operation on the same queue.
3. By performing a continue operation on a queue. The process performing the continue operation automatically returns from its monitor procedure. If another process is waiting in the queue, that process will immediately resume the execution of the monitor procedure in which it was delayed.

A monitor queue is either empty or holds a single process. A multiprocess queue can be implemented as an array of single-process queues.

In October 1973 Ole-Johan Dahl suggested to Tony Hoare that a continue operation should terminate a monitor call (Hoare 1974a; McKeag 1991). Hoare may have told me about this idea during his visit to Caltech in January 1974.

In the pipeline example a *printer process* is defined by a *process type* (Algorithm 6). A process parameter defines the only monitor (a line buffer) that is accessible to the process.

The pipeline program initializes a printer process as follows:

```
type printerprocess =
process(buffer: linebuffer);
var param: ioparam; text: line;
begin
  param.operation := output;
  cycle
    buffer.receive(text);
    repeat io(text, param, printdevice)
    until param.status = complete
  end
end
```

Algorithm 6 A process type.

```
var outbuffer: linebuffer; writer: printerprocess;

init writer(outbuffer)
```

The *init* statement allocates memory for fresh instances of the local variables declared at the beginning of the process type and starts execution of the process.

The Concurrent Pascal compiler ensures that the local variables of a process are inaccessible to other processes (and monitors). It also checks that a printer process uses its own line buffer only.

A printer process repeats the same *cycle* of operations endlessly. In each cycle the process receives a line from the buffer and prints it. The standard procedure *io* delays the process until the line has been output (or the printing has failed). In this simple example, the printing is repeated until it has been successfully completed.

All input/output are indivisible operations that hide peripheral interrupts. Consequently a process and a peripheral device cannot access the same variable simultaneously.

The complete *pipeline program* defines a parameterless process known as the *initial process*. This process includes definitions of all the monitor and process types used by the pipeline (Algorithm 7).

The execution of the program activates a single initial process, which then initializes two buffer monitors and activates three concurrent processes (by means of an *init* statement).

In addition to processes and monitors, Concurrent Pascal also includes

```
type linebuffer =
monitor
  . . .
end;

type cardprocess =
process(buffer: linebuffer);
  . . .
end;

type copyprocess =
process(inbuffer, outbuffer: linebuffer);
  . . .
end;

type printerprocess =
process(buffer: linebuffer);
  . . .
end;

var inbuffer, outbuffer: linebuffer;
    reader: cardprocess;
    copier: copyprocess;
    writer: printerprocess;
begin
  init inbuffer, outbuffer,
       reader(inbuffer),
       copier(inbuffer, outbuffer),
       writer(outbuffer)
end.
```

Algorithm 7 A program.

classes. A class is a module that cannot be called simultaneously by processes. It must be local to a single process, monitor or class.

Algorithm 8 shows a *class type*. A module of this type has access to a single line buffer. The class procedure extends a line with a left margin of 26 spaces and terminates it with a newline character before sending the line through the buffer.

```
type linemaker =
class(buffer: linebuffer);
var image: line; charno: integer;

procedure entry write(text: line);
begin
  for charno := 27 to 106 do
    image[charno] := text[charno−26];
  buffer.send(image)
end;

begin
  for charno := 1 to 26 do
    image[charno] := space;
  image[107] := newline
end
```

Algorithm 8 A class type.

A Simula program can bypass the procedures of a class and change the class variables in ways that are incompatible with the function of the class. This loophole was removed in Concurrent Pascal. A variable declared within a class can be read (but not changed) outside the class, provided the variable is prefixed with the word *entry*. Entry variables are not permitted in monitors.

Table 1 shows how Concurrent Pascal differs from Pascal. It lists the features that were added to Pascal as well as those that were excluded.

I have already illustrated the major concepts of Concurrent Pascal: processes, monitors, classes, and queues, as well as init and cycle statements (Algorithms 5–8).

The programming of terminal and printer drivers is supported by a notation for *control characters*. The following example

Table 1 Concurrent Pascal versus Pascal.

Added features	Excluded features
process types	file types
monitor types	pointer types
class types	packed arrays
queues	variant records
init statements	goto statements
cycle statements	recursion
control characters	
universal parameters	
program declarations	

```
const formfeed = '(:12:)'
```

defines *form feed* as ASCII character number 12.

When you program a procedure that reads a disk page, you cannot anticipate all the possible data types that users will assign to this page in the future. This is one of the few cases in which one cannot hide machine detail.

Concurrent Pascal uses *universal parameters* to relax type checking in device procedures. In the following procedure declaration

```
type diskpage = array [1..256] of integer;

procedure readdisk(pageno: integer;
  var page: univ diskpage);
begin . . . end
```

the key word *univ* indicates that the procedure may be called with any argument that has the same length as an array of 256 integers. The type checking is relaxed only at the point where the procedure is called. No variable is treated as a typeless bit pattern throughout a program (Brinch Hansen 1975d).

A *program declaration* enables a Concurrent Pascal program to call a sequential user (or system) program written in a subset of Pascal. The program declaration includes a list of procedures that the Pascal program may call. The details of this *ad hoc* mechanism are described in the Concurrent Pascal report (Brinch Hansen 1975a).

Since an operating system written in Concurrent Pascal must implement its own filing system, *file types* cannot be built into the language.

Pointer types were excluded to prevent a process from obtaining unsynchronized access to a variable of another process through a pointer transmitted through a monitor. In the absence of pointers, processes can access shared variables through monitor procedures only.

Packed arrays, *variant records*, and *goto statements* were eliminated to simplify the language.

Later I will explain my reasons for eliminating *recursive procedures and functions.*

The complete syntax and semantics of Concurrent Pascal are defined in the language report (Brinch Hansen 1975a).

Concurrent Pascal was designed according to the principles discussed earlier. It is a programming language that supports modular programming with processes, monitors, and classes. The syntax clearly shows that each module consists of a set of variables, a set of procedures, and an initial statement. Each module defines the representation and possible transformations of a data structure. A module cannot access the variables of another module. This simple scope rule enables a compiler to detect race conditions before a program is executed. The automatic synchronization of monitor calls prevents other race conditions at run time.

The programming tricks of assembly language are impossible in Concurrent Pascal: there are no typeless memory words, registers, and addresses in the language. The programmer is not even aware of the existence of physical processors and interrupts. The language is so secure that concurrent processes run without any form of memory protection!

My working habits unfortunately make it impossible for me to remember the alternative forms of syntax, scope, and type rules that I must have considered while designing the language. I evaluate language concepts by using them for program design. I develop a program by writing numerous drafts of the program text. A draft is immediately rejected if it is not in some way simpler and more elegant than the previous one. An improved draft immediately replaces the previous one, which is thrown in the waste basket. Otherwise I would drown in paper and half-baked ideas. As I jump from one draft to another without slowing myself down, a beautiful design eventually emerges. When that happens, I write a simple description of the program and rewrite it one more time using the same terminology as in the description. By then I have already forgotten most of the alternatives. And, twenty years later, I don't remember any of them.

The translation problem

An early six-pass compiler was never released. Although it worked perfectly, I found it too complicated. Each pass was written by a different student who had difficulty understanding the rest of the compiler.

From June through September 1974 my student, Al Hartmann, wrote another Concurrent Pascal compiler. His goal was to be able to compile small operating systems on a PDP 11/45 minicomputer with at least 32 k bytes of memory and a slow, removable disk. The compiler was divided into seven passes to fit into a small memory. It consisted of 8300 lines written in Pascal and could be completely understood by one person. Systematic testing of the compiler took three months, from October through December 1974.

The Concurrent Pascal compiler was used from January 1975 without problems. It was described in the Ph.D. thesis (Hartmann 1975), later published as a monograph.

In another month Al Hartmann derived a compiler for a Pascal subset known as *Sequential Pascal* (Brinch Hansen 1975b). It compiled the largest pass of the Concurrent Pascal compiler in 3 min. The compilation speed was limited mostly by the disk.

When we say that a program is concurrent, we are really talking about its behavior at run time. During compilation a program written in any language is just a piece of text, which is checked for correct syntax, scope of declarations, and types of operands. Consequently, the compilation of processes, monitors, and classes in Concurrent Pascal is very similar to the compilation of data types and procedures in Sequential Pascal.

The art of compromise

The Concurrent Pascal compiler generated code for a simple machine tailored to the language. I borrowed this idea from a portable Pascal compiler (Nori 1974). My main concern was to simplify code generation. The portability of Concurrent Pascal was just a useful by-product of this decision.

The Concurrent Pascal machine was simulated by a kernel of 8 k bytes written in assembly language. The kernel multiplexed a PDP 11/45 processor among concurrent processes and executed them using a technique known as *threaded code* (Bell 1973). It also performed basic input/output from a fixed set of peripherals (terminal, disk, magnetic tape, line printer, and card reader).

I wrote the kernel in Pascal extended with classes. Robert Deverill and Tom Zepko translated the kernel into assembly language. It was completed in January 1975 and described in a report (Brinch Hansen 1975e).

I made major compromises to make program execution as efficient as possible:

- All procedures must be non-recursive. This rule imposes a strict hierarchical structure on processes and monitors that prevents monitor deadlocks.
- All processes, monitors, and classes exist forever. This is acceptable in operating systems and real-time systems that perform a fixed number of tasks forever.
- All processes and monitors must be activated by the initial process.

These compromises made memory allocation trivivial. The first rule enabled the compiler to determine the memory requirements of each module. The first two rules made static memory allocation possible. The third rule made it possible to combine the kernel, the program code, and all monitor variables into a single memory segment that was included in the address space of every process. This prevented fragmentation of a limited address space and made monitor calls almost as fast as simple procedure calls.

By putting simplicity and efficiency first we undoubtedly lost generality. But the psychological effect of these compromises was phenomenal. Suddenly an overwhelming task seemed manageable.

Fifteen years later, I realized that the severe restrictions of Concurrent Pascal had made it impossible for me to discover and appreciate the powerful concept of recursive processes (Brinch Hansen 1989a, 1989b).

Learning to program again

After defining Concurrent Pascal, I wrote a series of model operating systems to evaluate the language. The new language had a dramatic (and unexpected) impact on my style of programming.

It was the first time I had programmed in a language that enabled me to divide programs into modules that could be programmed and tested separately. The creative part was clearly the initial selection of modules and the combination of modules into hierarchical structures. The programming of each module was often trivial. I soon adopted the rule that each module should consist of no more than one page of text. This discipline made programs far more readable and reliable than traditional programs that operate on global data structures.

The first operating system written in Concurrent Pascal (called *Deamy*) was used only to evaluate the expressive power of the language and was never built (Brinch Hansen 1974b). The second one (called *Pilot*) was used for several months but was too slow.

In May 1975 I finished the *Solo* system, a single-user operating system for the development of Concurrent and Sequential Pascal programs on a PDP 11/45. The operating system was written in Concurrent Pascal. All other programs, including the Concurrent and Sequential Pascal compilers, were written in Sequential Pascal. The heart of Solo was a job process that compiled and ran programs stored on a disk. Two additional processes performed input and output simultaneously. System commands enabled the user to replace Solo with any other Concurrent Pascal program stored on disk, or to restart Solo again. Al Hartmann had already written the compilers. I wrote the operating system and its utility programs in three months. Wolfgang Franzen measured and improved the performance of the disk allocation algorithm.

Solo was the first major example of a concurrent program consisting of processes, monitors, and classes (Brinch Hansen 1975c).

At Regnecentralen I had been involved in the design of process control programs for a chemical plant, a power plant, and a weather bureau. These real-time applications had one thing in common: each was unique in its software requirements. Consequently the programs were expensive to develop.

When the cost of a large program cannot be shared by many users, the only practical way of reducing cost is to give process control engineers a high-level language for concurrent programming. I illustrated this point by means of a real-time scheduler, which had been programmed in assembly language at Regnecentralen. I now reprogrammed the same scheduler in Concurrent Pascal.

The *real-time scheduler* executed a fixed number of task processes with frequencies chosen by an operator. I wrote it in three days. It took 3 hours of machine time to test it systematically. Writing a description took another couple of days. So the whole program was developed in less than a week (Brinch Hansen 1975f).

At the end of 1975 I wrote a *job-stream system* that compiled and executed short Pascal programs input from a card reader and output on a line printer. Input, execution, and output took place simultaneously using buffers stored on a disk. A user job was preempted if its compilation and execution time exceeded 1 minute. I designed, programmed, and tested the system in 10 days. When the system was finished, it ran short jobs continuously at the speed of the line printer (Brinch Hansen 1976a).

It was a pleasant surprise to discover that 14 modules from Solo could be used unchanged in the job stream system. This is the earliest example I know of different operating systems using the same modules.

Each model operating system was a Concurrent Pascal program of about 1000 lines of text divided into 15–25 modules. A module was roughly one page of text (50–60 lines) with about 5 procedures of 10–15 lines each (Table 2).

Table 2 Model operating systems.

	Solo	Job stream	Real time
Lines	1300	1400	600
Modules	23	24	13
Lines/module	57	58	46
Procedures/module	5	4	4
Lines/procedure	11	15	12

These examples showed that it was possible to build nontrivial concurrent programs from very simple modules that could be studied page by page (Brinch Hansen 1977a).

Compared to assembly language, Concurrent Pascal reduced my programming effort by an order of magnitude and made concurrent programs so simple that a journal could publish the entire text of a 1300 line program (Brinch Hansen 1975c).

The modules of a concurrent program were tested one at a time starting with those that did not depend on other modules. In each test run, the initial process was replaced by a short test process that called the top module and made it execute all its statements at least once. When a module worked, another one was tested on top of it. Detailed examples of how this was done are described in Brinch Hansen (1977b, 1978d).

Dijkstra (1967) had used a similar procedure to test the T.H.E. multiprogramming system, which was written in assembly language. Concurrent

Pascal made bottom-up testing secure. The compilation checks of access rights ensured that new (untested) modules did not make old (tested) modules fail. My experience was that a well-designed concurrent program of one thousand lines required a couple of compilations followed by one test run per module. And then it worked (Brinch Hansen 1977a).

The end of the beginning

In July 1976 I joined the University of Southern California as Professor and Chairman of Computer Science. I also finished a book on the new programming methodology entitled *The Architecture of Concurrent Programs* (Brinch Hansen 1977b).

My research on Concurrent Pascal was now entering its final phase. I wrote my last Concurrent Pascal program: a message router for a *ring network* of PDP 11/45 computers. I proved that it was deadlock-free and would deliver all messages within a finite time. The ideas of this program were developed in discussions with B. Heidebrecht, D. Heimbigner, F. Stepczyk, and R. Vossler at TRW Systems (Brinch Hansen 1977c).

My Ph.D. student, Jørgen Staunstrup, and I introduced *transition commands*—a formal notation for specifying process synchronization as state transitions (Brinch Hansen 1978a). In his Ph.D. thesis, Staunstrup (1978) used this tool to specify major parts of the Solo system.

Another of my Ph.D. students, Jon Fellows, wrote one more operating system in Concurrent Pascal: the *Trio* system, which enabled users to simultaneously develop and execute programs on a PDP 11/55 minicomputer with three terminals and a memory of 160 k bytes. Jon Fellows was assisted in a few cases by Habib Maghami (Brinch Hansen 1980; Fellows 1980).

I now moved into another area that was little understood at the time: the programming of processes on a multicomputer without shared memory. I introduced the idea of a synchronized procedure that can be called by one process and executed by another process (Brinch Hansen 1978b). This proposal combined processes and monitors into a single concept, called *distributed processes.*

This communication paradigm is also known as *remote procedure calls.* I recently discovered that it was first proposed by Jim White (1976). However, White did not explain how to prevent race conditions between unsynchronized remote calls and local processes, which are being executed by the same processor. This flaw potentially made remote procedure calls as unsafe as interrupts that cannot be disabled! Disaster was avoided by a programming convention: a process that handled a remote call immediately made a similar call to a local monitor (Lynch 1991). In other words, insecure remote procedure calls were used only as an implementation technique for secure remote monitor calls.

My Ph.D. student, Charles Hayden (1979), implemented an experimental language with distributed processes on an LSI 11 and evaluated the new paradigm by writing small simulation programs.

According to Roubine (1980), my proposal was "a source of inspiration in the design of the Ada tasking facilities." The Ada *rendezvous* combines the remote procedure call of distributed processes with the selection of alternative interactions in communicating sequential processes (Hoare 1978).

My keynote address on concurrent programming at the IEEE Computer Software and Applications Conference in Chicago, November 1978, concluded five years of experience with the first abstract programming language for operating system development (Brinch Hansen 1978c).

The *milestones* of the project were:

1974	Concurrent Pascal defined
	Concurrent Pascal implemented
1975	Concurrent Pascal paper
	Solo operating system
	Real-time scheduler
	Job-stream system
1976	Solo papers
	System distribution
1977	*The Architecture of Concurrent Programs*
	Ring network
1978	Trio operating system
	Distributed processes

In Brinch Hansen (1980), Jon Fellows and I concluded that

> The underlying concepts of processes, monitors and classes can now be regarded as proven tools for software engineering. So it is time to do something else.

Feedback

Concurrent Pascal and Solo have been assessed by a number of computer scientists.

In a paper on programming languages for real-time control, C.A.R. Hoare (1976) summarized Concurrent Pascal:

> This is one of the few successful extensions of Pascal, and includes well structured capabilities for parallel processing, for exclusion and for synchronization. It was tested before publication in the construction of a small operating system, which promises well for its suitability for real-time programming. Although it does not claim to offer a final solution of the problem it tackles, it is an outstanding example of the best of academic research in this area.

In a detailed assessment of Concurrent Pascal, D. Coleman (1980) wrote:

> The process, monitor and class concepts work equally well for application and system programs. Therefore in that respect the language works admirably. However, because the language is meant for operating systems, all programs run on the bare Pascal machine and every application program must contain modules to provide facilities normally provided by the operating system, e.g. to access the file store.

P.W. Abrahams (1978) found that the modularity of the model operating systems definitely contributed to their readability. However,

> Since the programs are always referring to entities defined earlier, and since these entities are often quite similar, I found that a good deal of page flipping was in fact necessary.

In a review of *The Architecture of Concurrent Programs*, R.A. Maddux and H. Mills (1979) wrote: “This is, as far as we know, the first book published on concurrent programming.” They were particularly pleased with the Solo system:

> Here, an entire operating system is visible, with every line of program open to scrutiny. There is no hidden mystery, and after studying such extensive examples, the reader feels that he could tackle similar jobs and that he could change the system at will. Never before have we seen an operating system shown in such detail and in a manner so amenable to modification.

In a survey paper on Concurrent Programming, R.E. Bryant and J.B. Dennis (1979) found that

> The ability to write an operating system in a high level language, including the communication and synchronization between processes, is an important advance in concurrent programming.

A final remark by D. Coleman (1980):

> Concurrent Pascal's main achievement is that it shows how much can be achieved by a simple language that utilises compile time checking to the maximum. It will be a great pity if future language designers do not adhere to these same two principles.

The limitations of the language will be discussed below.

4 Further Development

Since 1975 many other researchers have explored the use of Concurrent Pascal on a variety of computers.

Moving a language

At Caltech we prepared a distribution tape with the source text and portable code of the Solo system, including the Concurrent and Sequential Pascal compilers. The system reports were supplemented by implementation notes (Brinch Hansen 1976b).

By the spring of 1976 we had distributed the system to 75 companies and 100 universities in 21 countries: Australia, Austria, Belgium, Canada, Denmark, Finland, France, Germany, Great Britain, Holland, India, Ireland, Italy, Japan, Norway, South Africa, the Soviet Union, Spain, Sweden, Switzerland, and the United States.

D. Neal and V. Wallentine (1978) moved Concurrent Pascal and Solo to an Interdata 8/32 minicomputer in four months and to an NCR 8250 in another two months. The biggest stumbling block was the addressing scheme of the PDP 11. They wrote:

> It is clear that a system requiring so little effort to be moved between vastly differing architectures must have been well designed from the outset. In addition, with a single exception (sets

and variants), all of the problem points were mentioned by the implementation notes accompanying the distributed system.

M.S. Powell (1979) and two students moved Concurrent Pascal to a Modular 1 in six months. Architectural differences between the source and target computers caused some portability problems. According to Powell,

> Brinch Hansen makes no claims about the portability of Solo, yet our experience shows that a system designed and documented this way can be moved fairly easily even when the target machine has a totally different architecture to that of the source machine.
>
> Since the system has been in use we have found it easy to use and simple to modify at both high and low levels.

S.E. Mattson (1980) moved Concurrent Pascal (without Solo) to an LSI 11 in four months. He found four errors in the compiler. He felt that

> The kernel is a rather complex program and although the assembly code was commented in a language that resembles Concurrent Pascal it was hard to understand in detail.
>
> The implementation is a tool of significant value for teaching, research, and engineering. It has been used with success in an undergraduate course.

J.M. Kerridge (1982) moved Concurrent Pascal to an IBM 370/145 in nine months part-time by rewriting the kernel in Fortran. He then moved it to a Honeywell system in one day! In his view

> The original software was extremely well documented and commented but there was still a large amount of 'hacking' which had to be undertaken before the system could be transported.

Concurrent Pascal was moved to many other computers (Löhr 1977; Bochmann 1979; Dunman 1982; Ravn 1982).

The limits of design

Several researchers described the experience of using Concurrent Pascal for system design.

A research group at TRW Systems used Concurrent Pascal for signal and image processing on a network of PDP 11/70s. Initially, the group had to extend the kernel with complicated device drivers written in assembly language. Later, D. Heimbigner (1978) redefined the *io* procedure and was able to program arbitrary device drivers in Concurrent Pascal (without extending the kernel).

N. Graef (1979) and others designed a small time-sharing system based on Solo with swapping of job processes. They described the performance as unsatisfactory compared to Unix.

After designing a multiterminal version of Solo, D. Coleman (1979) and others concluded that

> *writing minicomputer operating systems by using Concurrent Pascal to provide the framework of concurrency for Sequential Pascal utilities is only really suited to single user systems.*

G.V. Bochmann and T. Joachim (1979) implemented the X.25 communication protocol in Concurrent Pascal on a Xerox Sigma 6.

H.S.M. Kruijer (1982b) described a multiterminal system for transaction processing implemented by a Concurrent Pascal program of 2200 lines for a PDP 11/34. He wrote:

> The work described in this paper shows that Concurrent Pascal is suitable for the construction of medium-sized multi-user systems. It has been found that the application of techniques which aim at enhancing portability, namely the exclusion of low-level features from the language and their implementation in the form of a kernel simulating a virtual machine, does not prevent systems written in Concurrent Pascal from being efficient. Moreover, both the properties of the language (its simplicity, high level, dependence on syntax rules) and its facilities (especially those for modularization) greatly contribute to obtaining reliable and adaptable system software. To illustrate this point it is relevant to mention that for the Multi operating system, a number of modules of Solo have been used which together amount to about 700 lines of Concurrent Pascal. The use of these modules

> in a different context was accomplished without interfacing problems and revealed only one error in one of the modules. These observations are in sharp contrast to our experience with commercially available operating systems.

Kruijer (1982a) also discovered a single (but subtle) error in the Concurrent Pascal kernel.

P. Møller-Nielsen and J. Staunstrup (1984) summarized four years of experience with a multiprocessor programmed in Concurrent Pascal. They discussed parallel algorithms for quicksort, mergesort, root finding, and branch-and-bound optimization.

The static memory allocation of the Concurrent Pascal implementation made the language impractical for the design of larger operating systems. In Brinch Hansen (1977b) I pointed out that the language was never intended for that purpose:

> This book describes a range of small operating systems. Each of them provides a special service in the most efficient and simple manner. They show that Concurrent Pascal is a useful programming language for minicomputer operating systems and dedicated real-time applications. I expect that the language will be useful (but not sufficient) for writing large, general-purpose operating systems. But that still remains to be seen. I have tried to make a programming tool that is very convenient for many applications rather than one which is tolerable for all purposes.

Evolution of an idea

Concurrent Pascal was followed by more than a dozen *monitor languages* (Table 3). Some were inspired by Concurrent Pascal; others were developed independently, inspired by the monitor concept.

I will not attempt to discuss monitor languages that were developed after Concurrent Pascal. I hope that the designers of these languages will write personal histories of their own contributions. However, since I have not programmed in their languages, I cannot evaluate them or compare them with Concurrent Pascal.

Table 3 Monitor languages.

Language	Reference
Concurrent Pascal	Brinch Hansen (1974d)
Simone	Kaubisch (1976)
Modula	Wirth (1977)
CSP/k	Holt (1978)
CCNPascal	Narayana (1979)
PLY	Nehmer (1979)
Pascal Plus	Welsh (1979)
Mesa	Lampson (1980)
SB-Mod	Bernstein (1981)
Concurrent Euclid	Holt (1982)
Pascalc	Whiddett (1983)
Concurrent C	Tsujino (1984)
Emerald	Black (1986)
Real-time Euclid	Kligerman (1986)
Pascal-FC	Burns (1988)
Turing Plus	Holt (1988)
Predula	Ringström (1990)

Spreading the word

Monitors and monitor languages have been discussed in many survey papers and textbooks. The following list of publication dates gives an idea of how rapidly the monitor paradigm spread through the computer science community.

- *Survey papers*

  Brinch Hansen (1973d), Andrews (1977), Bryant (1979), Stotts (1982), Andrews (1983), Appelbe (1985), Bal (1989).

- *Operating systems texts*

  Brinch Hansen (1973b), Tsichritzis (1974), Peterson (1983), Deitel (1984), Janson (1985), Krakowiak (1988), Pinkert (1989), Nutt (1992), Tanenbaum (1992).

- *Concurrent programming texts*

  Brinch Hansen (1977b), Holt (1978), Welsh (1980), Ben-Ari (1982), Holt (1983), Andre (1985), Boyle (1987), Perrott (1987), Whiddett

(1987), Bustard (1988), Gehani (1988), Krishnamurthy (1989), Raynal (1990), Williams (1990), Andrews (1991).

- *Programming language texts*

  Turski (1978), Tennent (1981), Ghezzi (1982), Young (1982), Horowitz (1983a), Schneider (1984), Bishop (1986), Wilson (1988), Sebesta (1989).

- *Annotated bibliography*

  Bell (1983).

5 In Retrospect

It seems natural to end the story by expressing my own mixed feelings about monitors and Concurrent Pascal.

The neglected problems

Today I have strong reservations about the monitor concept. It is a very clever combination of shared variables, procedures, process scheduling, and modularity. It enabled us to solve problems that we would not have undertaken without a commitment to this paradigm. But, like most of our programming tools, it is somewhat baroque and lacks the elegance that comes from utter simplicity only.

The monitor concept has often been criticized on two grounds: the complex details of process scheduling and the issue of nested monitor calls.

As a language designer, I have always felt that one should experiment with the simplest possible ideas before adopting more complicated ones. This led me to use single-process queues and combine process continuation with monitor exit. I felt that the merits of a signaling scheme could be established only by designing real operating systems (but not by looking at small programming exercises). Since Concurrent Pascal was the first monitor language, I was unable to benefit from the practical experience of others. After designing small operating systems, I concluded that first-in, first-out queues are indeed more convenient to use.

In 1974, when I designed the language, the papers by Howard (1976a, 1976b) and Kessels (1977) on monitor signaling had not yet been published. In any case, the virtues of different signaling mechanisms still strike me as being only mildly interesting. In most cases, any one of them will do, and all of them (including my own) are somewhat complicated. Fortunately, monitors have the marvelous property of *hiding* the details of scheduling from concurrent processes.

In my first monitor paper (Brinch Hansen 1974c) I characterized *nested monitor calls* as a natural and desirable programming feature:

> A monitor can call shared procedures implemented within other monitors. This makes it possible to build an operating system as a *hierarchy of processes and monitors*.

If a process delays itself within a nested sequence of monitor calls, it releases access to the last monitor only, but leaves the previous monitors temporarily inaccessible to other processes. Lister (1977) felt that this situation might degrade performance or cause deadlock:

> The only implementation known to the author in which the nested call problem is tackled head-on, rather than being merely avoided, is that by Brinch Hansen (1975e). In this [Concurrent Pascal] implementation a local exclusion mechanism is used for each monitor, and a [delay] operation causes release of exclusion on only the most recently called monitor. It is not clear what measures, if any, are taken to avoid the degradation of performance and potential for deadlock mentioned earlier.

Lister (1977) offered no performance figures or program examples to prove the existence of such a problem. The hypothetical "problem" of nested monitor calls was discussed further by Haddon (1977), Keedy (1978), Wettstein (1978), and Kotulski (1987)—still without experimental evidence. In a paper on "The non-problem of nested monitor calls" Parnas (1978) finally declared that the problem was too vaguely formulated to be solvable.

Two years before this discussion started I had written three model operating systems in Concurrent Pascal. I used nested monitor calls in every one of them without any problems. These calls were a natural and inevitable consequence of the hierarchical program structures.

The discomfort of complexity

The monitor was undoubtedly a paradigm that for a time provided model problems and solutions to the computer science community. It may be argued that its proper role is to define a useful programming style, and that it is a mistake to include it in a programming language. To an engineer, this viewpoint has merits. To a scientist, it is less convincing.

When an idea is seen just as a programming style, programmers seldom define it precisely. They constantly bend the (unstated) rules of the game and mix it with other imprecise paradigms. This lack of rigor makes it rather difficult to explore the limits of a new idea.

I never considered Concurrent Pascal to be a final solution to anything. It was an experimental tool that imposed an intellectual discipline on me. By embedding monitors in a programming language I committed myself to defining the concept and its relationship to processes concisely. I deliberately made monitors the only communication mechanism in the language to ensure that we would discover the limitations of the concept.

Concurrent Pascal was the first programming language I designed. From my present perspective, it has all the flaws that are inevitable in a first venture.

In a later essay on language description (Brinch Hansen 1981), I wrote:

> The task of writing a language report that explains a programming language with complete clarity to its implementors and users may look deceptively easy to someone who hasn't done it before. But in reality it is one the most difficult intellectual tasks in the field of programming.

Well, I was someone who hadn't done it before, and the Concurrent Pascal report suffered from all the problems I mentioned in the essay.

I am particularly uncomfortable with the many *ad hoc* restrictions in the language. For example,

- Module types cannot be defined within procedures.
- Procedures cannot be defined within procedures.
- Module instances cannot be declared within procedures.
- Queues can only be declared as global variables of monitor types.
- Queues cannot be parameters of procedure entries.
- Process instances can only be declared in the initial process.
- A module type cannot refer to the variables of another module type.

- A module type cannot call its own procedure entries.
- A procedure cannot call itself.
- A continue operation can only be performed within a monitor procedure entry.
- Assignments cannot be performed on variables of type module or queue.

These rules were carefully chosen to make the language secure and enforce the compromises discussed earlier. But they all *restrict* the *generality* of the language concepts and the ways in which they may be combined.

There are about twenty rules of this kind in Concurrent Pascal (Brinch Hansen 1975a). I will spare you the rest. They are an unmistakable symptom of complexity.

After Concurrent Pascal I developed two smaller languages. Each of them was again designed to explore a single programming concept: conditional critical regions in *Edison*, and synchronous communication in *Joyce* (Brinch Hansen 1981, 1989a).

There are exactly three *ad hoc* restrictions in Joyce:

- A process cannot access global variables.
- A message cannot include a channel reference.
- Two processes cannot communicate by polling the same channel(s).

I think only the first one is really necessary.

Inventing the future

What am I most proud of? The answer is simple: We did something that had not been done before! We demonstrated that it is possible to write nontrivial concurrent programs exclusively in a secure programming language.

The particular paradigm we chose (monitors) was a detail only. The important thing was to discover if it was possible to add a new dimension to programming languages: *modular concurrency*.

Every revolution in programming language technology introduces abstract programming concepts for a new application domain. Fortran and Algol 60 were the first abstract languages for numerical computation. Pascal was used to implement its own compiler. Simula 67 introduced the class concept for simulation.

Before Concurrent Pascal it was not known whether operating systems could be written in secure programming languages without machine-dependent features. The discovery that this was indeed possible for small operating systems and real-time systems was far more important (I think) than the introduction of monitors.

Monitors made process communication abstract and secure. That was, of course, a breakthrough in the art of concurrent programming. However, the monitor concept was a detail in the sense that it was only one possible solution to the problem of making communication secure. Today we have three major communication paradigms: monitors, remote procedures, and message passing.

The development of secure language concepts for concurrent programming started in 1971. Fifteen years later Judy Bishop (1986) concluded:

> It is evident that the realm of concurrency is now firmly within the ambit of reliable languages and that future designs will provide for concurrent processing as a matter of course.

In the first survey paper on concurrent programming I cited 11 papers only, written by four researchers. None of them described a concurrent programming language (Brinch Hansen 1973d). The development of monitors and Concurrent Pascal started a wave of research in concurrent programming languages that still continues. A recent survey of the field lists over 200 references to nearly 100 languages (Bal 1989).

I don't think we have found the right programming concepts for parallel computers yet. When we do, they will almost certainly be very different from anything we know today.

Acknowledgements

This paper is dedicated to my former students who contributed to the Concurrent Pascal project:

Jon Fellows	Charles Hayden	Jørgen Staunstrup
Wolfgang Franzen	Habib Maghami	Tom Zepko
Al Hartmann		

I thank the following 60 colleagues for their helpful comments on earlier drafts of this paper:

Birger Andersen
Greg Andrews
Bill Atwood
Art Bernstein
Jean Bezivin
Judy Bishop
Coen Bron
Dave Bustard
Mani Chandy
Derek Coleman
Ole-Johan Dahl
Peter Denning
Jerry Feldman
Jon Fellows
Narain Gehani
Jonathan Greenfield
Al Hartmann
Charles Hayden
Dennis Heimbigner
John Hennessy
Tony Hoare
Ric Holt
Jim Horning
Giorgio Ingargiola
David Jefferson
Mathai Joseph
Eric Jul
Jon Kerridge
Don Knuth
Henk Kruijer
Andrew Lister
Bart Locanthi
Ewing Lusk
Bill Lynch
Rich McBride
Mike McKeag
Jan Madey
Roy Maddux
Mike Mahoney
Skip Mattson
Harlan Mills
Peter O'Hearn
Ross Overbeek
Niels Pedersen
Ron Perrott
Malcolm Powell
Brian Randell
Anders Ravn
Charles Reynolds
Johan Ringström
Bob Rosin
Fred Schneider
Avi Silberschatz
Jørgen Staunstrup
Wlad Turski
Virgil Wallentine
Peter Wegner
Dick Whiddett
Niklaus Wirth
Tom Zepko

I also thank the anonymous referees for their careful reviews of earlier drafts.

The Concurrent Pascal project was supported by the National Science Foundation under grant numbers DCR74-17331 and MCS77-05696, the design of Trio by the Army Research Office under contract number DAAG29-77-G-0192, and the development of Distributed Processes by the Office of Naval Research under contract numbers NR048-647 and NR049-415.

Appendix: Reviewers' Comments

In 1991 I sent earlier drafts of this paper to a number of computer scientists with a letter asking for their comments "with the understanding that I may quote your letter in the final paper." Many of their suggestions are incorporated in the revised paper. Here are some of their remaining remarks.

G. Andrews:

You claim that the particular paradigm you chose (monitors) was a . . . detail. The most important aspect of monitors is their role as a data encapsulation mechanism.

. . .

The contribution of Concurrent Pascal was indeed that it added a new dimension to programming languages: modular concurrency. Monitors (and classes) were essential to this contribution. And the modularization they introduced has greatly influenced most subsequent concurrent language proposals.

What is debateable about monitors are the details of synchronization, especially the signaling discipline.

. . .

I have not seen any radical new programming ideas emerge for several years now. Thus, I suspect that in the future the programming concepts we use for parallel computers will merely be refinements of things we know today.

D.W. Bustard:

The statement . . . "Today I have strong reservations about the monitor concept" tends to suggest that the *concept* is flawed. I don't agree. The basic concept of a data structure allowing processes exclusive access to its data still seems very important. What has never been handled satisfactorily, however, is the explicit queuing mechanism for process suspension and activation. I tinkered with several possibilities over a period of years but now (like Parnas) I feel that it would be better to give access to lower level facilities that allow users to implement a policy of their own liking. It is a mistake for language designers to treat potential users like children!

O.-J. Dahl:

I am grateful for your recognition of the role of the Simula 67 class concept; however, in the reference to it the name of my colleague Kristen Nygaard should occur along with mine . . . [Our] own historic paper, given at the ACM Conference on the "History of Programming Languages" . . . shows the extent to which either of us was dependent on the other in the discovery of the class concept.

. . .

I take issue with some of your reservations about Concurrent Pascal. Of course a language built around a small number of mechanisms used orthogonally is an ideal worth striving for. Still, when I read your 1977 book my reaction was that the art of imposing the right restrictions may be as important from an engineering point of view. So, here for once was a language, beautiful by its orthogonal design, which at the same time was the product of a competent engineer by the restrictions imposed in order to achieve implementation and execution efficiency. The adequacy of the language as a practical tool has been amply demonstrated.

P.J. Denning:

I had a love-hate relationship with monitors since first meeting them as "critical regions" in your 1973 book and then in Hoare's 1974 paper in the ACM Communications. What I loved about them was the way they brought together data abstraction (as we now call it) and synchronization. Suddenly we had a simple notation that allowed the expression of correct programs for the hard problems we faced constantly in operating systems design. What I hated about them was the need to understand the details of the queuing mechanism in order to understand how to use them. My students had to study carefully Hoare's notes on using semaphores to do the queueing. In this sense monitors had not broken away from the fine-grain mechanisms of semaphores.

I was therefore much interested in the next stages that you and Hoare reached, expressed in your 1978 papers in the ACM Communications. You had continued the line of development of monitors into distributed processes; Hoare had proposed communicating sequential processes, an approach motivated by the constraints of microprocessor design. I was more attracted to Hoare's proposal because of my own biases in thinking about how operating systems and parallel computers are actually built and how they manage work.

...

Even though in the end I found the monitor concept less to my liking than communicating processes, I still think that the monitor is a good idea, and that the observer it makes one of how operating systems work is a worthy observer to learn to be.

J.A. Feldman:

I was not personally involved with the [Concurrent Pascal] effort, but admired it and now find somewhat to my surprise that my current parallel Sather project relies on a version of monitors.

. . .

[It is] now clear that any large, scalable parallel machine will have physically distributed memory. There is a great deal of current research on hardware and software for uniform memory abstractions, but this seems to me unlikely to work. The structure of the programming language and code can provide crucial information on locality requirements so that the system doesn't need to do it all mindlessly. And that is where monitors come in.

Sather is an object-oriented language . . . The parallel constructs . . . are based on a primitive monitor type . . . [It] is remarkable that 20 years later the monitor concept is central to language developments well beyond the original conception.

J. Fellows:

Looking back at my studies at USC from 1978 to 1981, I can separate my thoughts into three areas: the concepts that underly monitors and classes, the language constructs that implement these concepts, and the quality of the demonstration programs that you (PBH) wrote. You have already addressed the first two in your paper. As for the third, I believe that the beauty of the structures you created using Concurrent Pascal created an aura of magical simplicity. While working with my own programs and those of other graduate students, I soon learned that ordinary, even ugly, programs could also be written in Concurrent Pascal . . . My current feeling is that the level of intellectual effort required to create a beautiful program structure cannot be reduced by programming language features, but that these features can more easily reveal a program's beauty to others who need to understand it.

. . .

The topic I chose to explore [in the Trio system] was the use of Concurrent Pascal's access restrictions to explicitly create a program access graph (or "uses" hierarchy between type instances) that achieved least privilege visibility between program components, meaning that no component has access to another component unless it is needed. For this purpose, I still believe that Concurrent Pascal's initialization-time binding of components is an improvement over the scope-based facilities of Modula, Edison, and Ada.

. . .

It is interesting to note that one of the most common complaints I heard (and made myself) was that classes should have been left in [Sequential

Pascal]. This would have extended many of the benefits available to system programmers to application programmers.

. . .

As I discovered when moving the compilers from Solo to Trio, there was a point at which the Operating System/Sequential Pascal interface was unsafe. As I recall, there was no type checking across the program invocation interface, which depended on correct hand-tailoring and consistent usage of the prefix for Sequential Pascal programs. In general, program invocation was the one operating system area that was not made transparently simple in Solo and Trio.

A.C. Hartmann:

There are really two histories interwoven in this paper—the history of the development of concurrent modular programming, and the history of one man's ruthless quest for simplicity in design and programming. The former topic is indifferent to whether one chooses to develop concurrency mechanisms for greater expressive power and more complex functionality, or, as you have chosen, to radically shorten and simplify the design of common concurrent systems. The Solo operating system is downright primitive in the sparseness of its features, representing a counter-cultural current against ever-increasing operating system complexity. Your style and taste in programming run almost counter to the second law of thermodynamics, that all closed systems tend towards increasing entropy and disorder.

In a world of Brinch Hansens (which may exist in some parallel dimension to ours), all systems tend towards reduced entropy over time and toward a blissful state of ultimate simplicity. Each new release of the operating system for one's personal workstation is smaller than the previous release, consumes fewer system resources, runs faster on simpler hardware, provides a reduced set of easier to use features than the last release, and carries a lower price tag. Hardware designers espousing the same philosophy produce successive single-chip microprocessors with exponentially declining transistor counts from generation to generation, dramatically shrinking die sizes, and reducing process steps by resorting to fewer, simpler device types. No one would need to "invent" RISC computing in this world, since reduced feature sets would be an inexorable law of nature.

. . .

Ironically the Concurrent Pascal compiler that I wrote was written in the language of its sister Sequential Pascal compiler, which had neither classes nor monitors. It was fifteen years later when I finally had access to a C++ system on a personal computer that I wrote my first modular program using abstract typing. To this day I have not written a concurrent program.

C.C. Hayden:

What was remarkable about [Concurrent Pascal] is that one could write experimental operating systems on a virtual machine without having to resort to machine registers, assembly language, etc. The development environment provided a way to do operating systems in a controlled way, on the "bare hardware" of a much nicer machine than any real computer...

I think the significance of the system was ... that one could provide a protected environment for concurrent programming—a high-level language environment which could maintain the illusion that there was no "machine" level. It was remarkable that through compile time restrictions and virtual machine error checking, that you could understand the program behavior by looking at the Pascal, not at the machine's registers and memory. It was remarkable that the machine could retain its integrity while programs were being developed, without hardware memory protection.

...

How has the monitor concept evolved? From my perspective, the concept of message passing between processes in disjoint address spaces was around before monitors, and has continued to dominate the monitor concept. The operating systems in most common use today have message passing paradigms. The Macintosh, Microsoft Windows, Unix running X windows: all force applications to be organized around an event loop, which receives an event message, unpacks it and dispatches to a handler, and carries out an action. These are just the "real time" system architecture of the 1960s. The monitor concept was an advance over the earlier message passing systems because it eliminated the event loop, message packing and unpacking, dispatching, etc. Concurrent Pascal hid all that mess, and made it possible to do it more efficiently by absorbing it into machine code or microcode, and eliminated the possibility of making errors. Why did it not become better accepted?

...

Maybe the problem monitors were meant to solve (concurrency in shared memory systems) was never really that important after all. The conventional wisdom is that concurrent systems cannot scale up if they share memory.

...

I have a deep respect for the monitor concept: in my opinion it is better than message passing, which is what we are stuck with. It is particularly powerful if used in the form of conditional critical regions. And I think the language Concurrent Pascal made a real advance in permitting easy experimentation with operating systems concepts and implementations. It allowed me to further my own education by building programs that I would not otherwise have been able to build. This taught me valuable lessons about programming styles and paradigms, about how important it is to be able to reason about programs when they cannot be reliably tested. Concurrent Pascal had to deal with such restrictive and peculiar hardware, almost unthinkably limiting by today's standards.

As your thinking evolved, the systems you built seemed to get smaller and more elegant, trying to achieve more generality and less complexity. This is a laudable goal for research languages, but I could never come to believe in it as applied to programming tools such as editors, formatters, etc. I know of few people who would want to adopt simpler tools ... Perhaps there is no longer any call for this kind of programming ... I am glad that I was able to educate myself before it was too late. The Concurrent Pascal system made that possible.

D. Heimbigner:

Concurrent Pascal is one of those languages that is very much under-appreciated. It was one of the first widely available languages to introduce both object-oriented concepts and concurrency (in the form of processes and monitors).

Concurrent Pascal is perhaps best known as one of the first languages to provide monitors as a synchronization device. Initially, I was a very strong believer in the monitor construct. After using the construct for a while, I recognized its flaws and was rather disenchanted with them. Since then, I have had some experience with Ada and its tasking model, and I am beginning to think that perhaps monitors were not such a bad concept after all.

I should also note that I am continually surprised at how long it is taking for concurrency constructs to become a standard part of every programmer's toolkit. The C and C++ communities, for example, are still arguing over a standard threads package. Most Unix kernels (except Mach) have no special provisions for handling threads, most Unix libraries are still not capable of working correctly in a parallel environment, and most Unix machines are still single processor. This seems to me to be a disgrace.

The concurrency elements of Concurrent Pascal were important, but I would also like to comment on its object-orientation. It was my first introduction to an object-oriented language. At the same time, (1976–1978), Smalltalk was mostly a rumor; it would be several years before it became available. Simula-67 was not widely available on any machine to which I had access.

So, when I encountered Concurrent Pascal, I spent a fair amount of time experimenting with its object-oriented constructs. As a result, I became a firm believer in that approach for programming and have continued to use the paradigm to this day.

It is interesting to compare Concurrent Pascal with, for example, Modula-2 and Ada. At one time, there was a discussion in the language community about the merits of objects (as represented in Smalltalk and Concurrent Pascal and Euclid) versus the merits of packages (as represented in Modula-2 and Ada). In retrospect, it seems amusing that these two concepts were considered comparable, rather than complementary. It also clear that the object point of view has prevailed (witness Modula-3 and Ada 9x).

J. Hennessy:

I had one interesting insight that I wanted to communicate to you. We have been experimenting with an object-oriented language (called Cool and based on C++) for programming parallel machines. The idea is to use the object structure as a basis for synchronization, dealing with data locality, and for implementing load balancing. Initially, we anticipated using a variety of synchronization primitives, including things such as futures, in addition to monitor-based constructs. Surprisingly, we found that the synchronization mechanisms based on monitors were adequate for most cases, and were much easier to implement (more efficient), and easier to understand. My advice is not to undersell monitors. I suspect that we will find that there are many more instances where this basic concept is useful!

C.A.R. Hoare:

I read your personal history with great enjoyment: it brings back with sharp clarity the excitement of our discussions at Marktoberdorf and Belfast in 1971. Even more valuably, it describes the whole history of a remarkably successful research engineering project, conducted with utmost regard for scientific integrity, and principles, which has enlarged the understanding of a whole generation of computing scientists and software engineers. That a subsequent generation has lost the understanding could be explained in another, much sadder, paper.

My only serious debate with your account is with the very last sentence. I do not believe that there is any "right" collection of programming concepts for parallel (or even sequential) computers. The design of a language is always a compromise, in which the good designer must take into account the desired level of abstraction, the target machine architecture, and the proposed range of applications. I therefore believe that the monitor concept will continue to be highly appropriate for implementation of operating systems on shared-store multi-processors. Of course, it will improve and adapt; its successful evolution is now the responsibility of those who follow your footsteps. Your full account of the original voyage of exploration will continue to inform, guide, and inspire them.

G. Ingargiola:

Your paper is faithful to what I remember.

You had this tremendous clarity about what you were doing in concurrency and languages; you made restrictive choices usually on the basis of efficiency (you list a number of such choices in your paper). You stated something like "start with as few and simple mechanisms as possible; add later only if it becomes necessary."

At least in your discussions and lectures, you built programs from english statements, making explicit the invariants and refining these statements, usually not modifying them, until the program was done.

I was amazed at how slowly you developed code when lecturing, and, by contrast, how fast you got debugged running code for the Concurrent Pascal compiler, and for various concurrent programs and the Solo OS.

You had very little interest in computer science topics outside of the area in which you were doing research. You made polite noises, you indicated interest, but your span of attention was minimal ...

The personnel involved in the Concurrent Pascal implementation is as small as you say ... Deverill contributed with his knowledge of the PDP 11 architecture and of its assembly language. Hartmann and you did the work. ...

I remember your excitement with the notion of "threaded code;" if I remember correctly, you thought it was your own invention and found out only later that others found it before.

In 1977, on the phone, you told me that you were working on a model of distributed computing where processes could make synchronous calls across processors. When later I heard about remote procedure calls, I assumed it was a variation on what you had said.

M. Joseph:

[Your paper] made very interesting reading and it took me back to the exciting days of early 1975 when you lectured on Concurrent Pascal in Bombay!

We spent quite a lot of 1975 studying Concurrent Pascal and deciding whether and how it could be used for our multiprocessor operating system project ...

Our version of the language (which we called CCNPascal, both because of its antecedents in Concurrent Pascal and because it was the language for the Close-Coupled Network project) ... was implemented on a DEC-10 and generated code for the DEC-10, PDP-11 and TDC-16 (and later a group produced a code generator for the Intel 8086). So perhaps it is fair to say that Concurrent Pascal had close ‘cousins’ on all of these machines!

I think there has been some general confusion about the role of Concurrent Pascal. On the one hand, it was used very successfully in the version that you supplied, by many people and for a variety of applications. On the other hand, the design of Concurrent Pascal also provided the springboard for people (like us) to make use of its concepts for designing larger languages which were applied to fairly ambitious tasks. So the monitor concept was fairly rugged and stood up well to the test of being used for large applications, and this is something that is not widely known.

Moreover, it was a language for which high quality code could be generated (something that implementors of Ada still aspire to). We had multi-pass cross-compiling versions of our compiler which generated extremely tight code and I later produced a one-pass version of the compiler which did a lot of on-the-fly optimization and produced PDP-11 code ...

With interest returning to shared memory multiprocessors, it seems quite appropriate that people should be reminded of the achievements of Concurrent Pascal.

J.M. Kerridge:

One of the reasons that I acquired Concurrent Pascal was to enable access to a Pascal system which at the time (1978) was the only way it could be made accessible on our IBM 370. It had the added benefit of introducing me to concurrent programming. This has lead me to continue working in the area of parallel systems allowing me to build highly parallel database machines based around the transputer and occam.

In this respect I find your comments . . . concerning the compromises that were made to effect efficient processing surprising. In the transputer/occam combination the same limitations have, to a large extent, also been imposed. This enables compile time checking of memory allocation and process interaction, which is vital for real-time embedded control systems. It is interesting that this too was the application environment from which you came originally.

If we consider the use of Ada for such safety-critical real-time systems then we have to use Safe Ada, which has exactly the same limitations. The full capability of Ada is only available with a large run-time support system about which it is impossible to reason!

Given the above points I believe that you have been somewhat hard in criticising Concurrent Pascal . . . Hindsight is a valuable tool especially after nearly 20 years! Many of the restrictions were reasonable given that you were experimenting with concurrency and not constructing a sequential language. Keeping things simple is a good axiom and though it is useful to have nested procedure declarations, as an example, it was not fundamental to the needs of concurrency experimentation. If many of these restrictions had been relaxed then Concurrent Pascal may never have seen the light of day.

H.S.M. Kruijer:

I (continue to) regard the specification and implementation of Concurrent Pascal as an impressive piece of work, combining the best results of Computer Science and making them available in the area of Software Engineering. More specifically, I regard this work and the publications on it as large-scale examples of the application of sound (computer) science resulting into high-quality "real-life" (software) engineering products, which still serve as a yardstick and a source of inspiration not only for (computer) scientists but also (more importantly) for practicising (software) engineers . . .

The use of Concurrent Pascal has played a major organising and professionalising role in the Computer science & Software engineering section of our Mathematics and Systems Engineering department. My paper in "Software—Practice and Experience" (1982) described a project carried out in the period 1976–1980, but other work has been done that has not been published:

—We have extended, during 1981–1983, the prototype application system referred to in my paper (namely a multi-user system for order taking and stock updating suited to Shell's Marketing (Sales) business) so as to run on a number of PDP 11 computers coupled via one common, shared communication channel (eventually an Ethernet). Therefore the Multi operating system described was transformed systematically into a distributed version, using remote procedure calls and client-server mechanisms and using Concurrent Pascal for the implementation of the data communication software needed.

. . .

—A prototype data acquisition system (for Shell's process control system in refineries and chemical plants) has been developed during 1981–1985, using Concurrent Pascal as implementation language.

—A data acquisition system for our Materials Research department has been developed during 1986–1989, using Concurrent Pascal for specification and design and using DEC hardware and software for implementation.

E.L. Lusk:

Our group's adoption of monitors as a central theme in our parallel programming work did not arise from an interest in elegant operating systems; it was absolutely forced on us by the task of writing application programs for real parallel processors. In 1983 Los Alamos obtained a Denelcor HEP, in many respects the first commercially available multiprocessor. Several different groups at Argonne tried it out, in 1984 Argonne got one too, and that facility evolved into Argonne's Advanced Computing Research Facility.

The HEP was programmed in a dialect of Fortran that allowed direct access to the full/empty bits in memory as a way of allowing ordinary program variables to be used for a kind of dataflow synchronization. The mechanism was efficient but dangerous. Ross Overbeek and I chose as our first project the parallel implementation of the unification algorithm from theorem proving. We found this a humbling experience, to say the least. While our colleagues proceeded smoothly with parallel versions of regular numerical algorithms, we suffered every type of bug associated with parallel algorithms. Finally we realized that we would have to retreat to intellectually higher ground. For the shared-memory computational model, monitors represented an abstraction that could be understood, reasoned about, and efficiently implemented. We used the HEP constructs to implement locks, used the locks to construct the basic monitor-building primitives, used these to build (portable, now, at this level) a library of useful monitors, and our problems disappeared for good. The macro package for the HEP has evolved through several generations, and its descendants are widely used for programming nearly all current shared-memory machines in C and Fortran.

...

I believe that the simple domain-composition algorithms that pay the freight for the current generation of massively-parallel machines are distracting many current users from realizing the long-term validity of the shared-memory model. Fast communication speeds do not make the shared memory irrelevant; it is the programming model that is important. Although the programming model for message passing is now relatively stable ..., no such consensus has arisen for the shared memory model ... In the long run monitors will be seen as the most useful paradigm for expressing algorithms for the shared-memory model. The shared-memory model, in turn, will return to greater prominence as more complex algorithms are moved to parallel computers.

Monitors shall arise again!

W.C. Lynch:

I think that monitors may have achieved more contemporary success than you might believe. In one sense I think of your description as one of the birth pains of an idea that has matured and stood the test of time.

...

In 1977 Xerox PARC/SDD [had] to construct a real time ... operating system (subsequently called "Pilot"). A large part of the problem was the specification of facilities for concurrent operation. The input experiences were ... 1) my experience in the design and implementation 1970–71 of Chios utilizing light-weight processes and Dijkstra PV operators ..., 2) Butler Lampson's proposal to incorporate Monitors and Condition variables, as described by Hoare, into Mesa and hence into Pilot ..., 3) the pre-disposition of management to leverage their experience with message passing paradyms in the SDS-XDS operating systems.

I was the convenor of this task force. Among others, Butler Lampson, Dave Redell, and Hugh Lauer were participants. Roger Needham was an occasional consultant.

Inputs 1) and 2) quickly converged, supported by the reality of your previous experience with Concurrent Pascal, but 3) led to a contentious stalemate, with each party claiming some inherent superiority over the other. This was finally resolved by the arguement presented in (Lauer 1978) which demonstrated that the views were equivalent in the sense that each could be executed in terms of the other. Since the intention was to combine support for concurrent processing with the benefits of Mesa, it was clear that the procedural view of Monitors was most compatible with the procedural language Mesa.

The design that resulted, smoothly incorporating threads (nee light weight processes), monitors, and condition variables into Mesa as built-in types, was eventually described in (Lampson 1980).

Lampson, Redell, and others moved on to DEC SRC and continued their work there. With the work on and introduction of the object oriented Modula-3, it was realized that a class structure allows the above threads related types to be implemented as a library without being implemented in the language . . .

Today threads libraries, a direct linear descendent of monitors, are de rigeur in the Unix world. I would say that the ideas created in the process that you describe are still a major force today.

. . .

I don't know what more one could ask in the way of ultimate triumph for an idea.

R.A. Overbeek:

In the early 1980's, E. Lusk and I were offered the opportunity of developing applications for a new parallel processor, the Denelcor HEP. Our application area was automated deduction, and our background in parallel computation was quite limited. In our first experiments, we worked with the programming constructs offered by Denelcor Fortran, which were low-level synchronization constructs. It became immediately apparent that we needed to develop higher-level, portable constructs. Our central source of both ideas and implementation guidelines during that period was Concurrent Pascal and your work that gave detailed implementation information. We developed a rather primitive set of tools and began work on several applications.

A year later, we were faced with moving our applications to several other machines. To our delight, we were able to port a 50,000 line implementation of a parallel logic programming engine from the HEP to a new Sequent Balance in just 4 hours. We went on to port the code to a variety of other shared-memory multiprocessors, and the benefits of portable constructs were quite apparent. Indeed, the ability to develop programs on machines in which the environment was relatively stable and for which adequate performance monitoring and debugging tools existed (most notably, the Sequent machines) and then move them to a number of "production" environments was extremely useful.

Later, we shifted our programming paradigm to include message-passing constructs. It is a tenable position that there are relatively few applications that benefit substantially from parallel processing, and that a majority of these can be formulated in ways that allow effective use of parallelism with message-passing constructs; that is, they do not require the capabilities we built into our earlier tools based on monitors. Furthermore, the ability to port applications based on message-passing to platforms like multicomputers or clusters of workstations is really quite attractive. I consider this a far from settled issue, but I have tentatively adopted this position.

Our work based on developing portable tools for exploring the potential exploitation of parallelism on the wide variety of machines that appeared in the 1980s benefited directly from the pioneering work on monitors. While we were never fortunate enough to be directly involved with the individuals that drove that effort, we did gradually come to grasp some of the issues that they had clarified.

N. Holm Pedersen:

It is with a feeling of nostalgia that we, at Brüel & Kjœr, read about the emerging of the ideas on which we have based most of the programming of our instruments for the last decade. We are still using the Concurrent Pascal (CP) language in full scale i.e. just finishing 2 instruments, each with programs of more than 1 Mbyte code written in Concurrent Pascal.

It is remarkable that the ideas of Concurrent Pascal is having a major effect on modern blockstructured languages such as Ada and C++. Excluding the monitor concept, Ada has inherited the structures of CP, and C++ has reinvented the Class-type. It caused some confusion in our company to hear C++ being named "the invention of the century" as we have created objects since 1980 in the form of Monitors and Classes. We are in the process of discovering that the freedom (nonhierarchical nature) of C++ is very nice but dangerous . . .

M.S. Powell:

I read the first draft of your paper "Monitors and Concurrent Pascal: A Personal History" with great interest. The work I did with Concurrent Pascal and the Solo system took place near the beginning of my academic career and much of my subsequent work has been strongly influenced by it. Your paper fills in many gaps in my knowledge of the history of the development of the underlying ideas. A number of things I did with Concurrent Pascal and Solo which may be of interest, but have never been published, are described below.

The characteristics of Solo which made it easy to port to a machine with a very different architecture to the PDP-11, also made it very easy to change and extend for practical and experimental purposes. The final configuration supported on Modular 1 hardware at UMIST ran across three processors with the file store distributed across two 28M byte exchangeable disk drives shared between the three processors. In this form the system supported many final year project students and research projects. Many compiler and language extensions were introduced, e.g. the Concurrent Pascal compiler was modified to support generic classes and the compiler and virtual machine were modified to support message passing through inter-process channels. A system which ran up to two passes of the compiler pipelined concurrently by two processors was also implemented.

. . .

After the Modular 1 system (around 1982) we moved onto a network of DEC LSI-11s connected together by a Cambridge ring. A distributed Concurrent Pascal implementation was constructed for this environment. During execution of the initial process, extensions to the virtual machine allowed processes and monitors to be assigned interactively to selected processors on the network. Monitor entry routine calls were implemented by remote procedure calls, and distribution was transparent to the concurrent program, i.e. we were able to run programs produced on the Modular 1 without recompilation.

. . .

A spin-off of my research work has been a system called Paradox which has been used to support teaching in the Computation Department for the last four years.

. . .

Inside the implementation of Paradox, unseen by most, Concurrent Pascal is alive and well at UMIST, and helping to support nearly 250 users every year.

A.P. Ravn:

I really enjoyed reading your paper on the history of Monitors and Concurrent Pascal. I shall refrain from commenting on who got the ideas for the monitor first; but I am sure that Concurrent Pascal was central for the dissemination of these ideas in software enginering.

. . .

The Concurrent Pascal system and its literature made it possible to combine theoretical concepts with experimental work. Probably the only way engineering can be taught.

It was a pleasure teaching courses based on Concurrent Pascal, and the students, who are now software developers, received a thorough knowledge of good system programming concepts. In some ways too good; when I meet them now, they find it hard to break away from these paradigms, even in distributed systems.

C.W. Reynolds:

There seem to me to be two central issues treated during the early period 1971–1973. First was the issue of medium term scheduling. How does a process wait for some condition to be true? . . .

It seems to me that the critical insight occurred in realizing that the responsibility for determining an awaited event has occurred must lie with the application programmer and not with the underlying run-time support. The awakening of processes awaiting events is part of the application algorithm and must be indicated by explicit announcement of the events by means of "signal" or "cause" commands present in the application algorithm.

This idea is clearly present as early as Brinch Hansen (1972b). Of less importance, but necessary to mention, is that there and in Concurrent Pascal, at most one process can be suspended in a single queue. Although this can be efficiently implemented and although it is possible to use it to simulate a queue containing multiple processes, the history of the last twenty years has shown the multiple process condition queue of Hoare (1974a) to be more popular.

The second central issue in this early period is the class notion from Simula. And there are two aspects to this. First is the encapsulation of procedures together with their shared variables and the prohibition of access to these shared variables by any procedures other than those encapsulated procedures. This notion of encapsulation appears in the unpublished draft Brinch Hansen (1971c) and it definitely appears in the textbook Brinch Hansen (1973b).

The second important aspect of the class concept is that a class is a mechanism for type definition so that multiple distinct variables of a class can be declared ... But, in the context of monitors, there is an important difference that appears in short-term scheduling of exclusive access to the monitor. Is this exclusion enforced for each class instance or is it enforced for the whole class at once? ... Mutual exclusion on individual instances of a class is possible in languages such as Concurrent Pascal and Mesa which adopted the Simula class style, whereas it is not possible in languages such as Pascal Plus, SP/k, Concurrent Euclid, Modula-2 and Ada which did not.

The treatment of monitors as Simula classes appears in the textbook Brinch Hansen (1973b) and is notably absent in Hoare (1974a).

I believe that another central issue treated but never resolved in this early period was the relationship between short-term scheduling in monitor access and medium term scheduling for awaited events. Evidence of this issue is found in the variety of signaling semantics proposed during this period. These included the Signal/Return semantics of Concurrent Pascal, the Signal/Unconditional-Wait semantics of Hoare (1974a) and the more prevalent Signal/Wait semantics of languages like SP/k and others. Multiple not-quite-satisfactory solutions to the problem indicate that it has not been resolved.

V. Wallentine:

Your comments on our experience porting Concurrent Pascal were completely accurate.

However, many things cannot be included in journal articles. The thrill of porting a useful language with such a small investment of time made it possible to use Concurrent Pascal in the academic environment. Many, many students were able to learn the concepts of concurrent programming and encapsulation using Concurrent Pascal. Having a concrete language to experiment with is essential to understanding the monitor concept. Using the concept of monitors as implemented in Concurrent C as a concrete example, they were able to better understand additional scheduling (and signaling) techniques.

I remember spending many hours with my research group discussing different signaling paradigms and hierarchical monitors. We also spent a significant amount of time implementing a distributed operating system (on top of Unix). This was a good test for the strength of the monitor concept.

T. Zepko:

Part of the history you describe is an important part of my own history.

At the time I was involved with Concurrent Pascal, I was an undergraduate and not so much concerned with the conceptual significance of the language as with learning how to build a language system from the ground up. I got the practical experience I wanted by working on the Concurrent Pascal compiler, the threaded code interpreter, and the operating system kernel. I have continued to do this same kind of work for the last fifteen years.

The concepts behind the Concurrent Pascal, the evolution of the ideas as you describe them, are clearer to me now than they were as a student. The needs you were addressing do require some years of experience to appreciate. But even as a student, some things left a lasting impression. What I learned from you, beyond specific programming techniques, is what I can only describe as a passion for clear thinking. This was obvious in the way you approached program design, and it was obviously the driving force behind the design of the Concurrent Pascal language.

. . .

Some of the ideas embodied in Concurrent Pascal were radical at the time. That they seem less so now is a tribute to the trailblazing nature of your work. Your approach to programming and to language design now has many advocates. Structured programming, modular design, strong typing, data encapsulation, and so on, are all considered essential elements of modern programming and have found their way into a wide variety of languages. I'm thankful to have played a part in this work.

References

Abrahams, P.W. 1978. Review of the Architecture of Concurrent Programs. *Computing Reviews 19*, (September).

Andre, F., Herman, D., and Verjus, J.-P. 1985. *Synchronization of Parallel Programs*, MIT Press, Cambridge, MA.

Andrews, G.R., and McGraw, J.R. 1977. Language features for process interaction. *SIGPLAN Notices 12*, (March), 114–127.

Andrews, G.R., and Schneider, F.B. 1983. Concepts and notations for concurrent programming. *ACM Computing Surveys 15*, 1 (January), 3–43. Reprinted in Gehani (1988), 3–69.

Andrews, G.R. 1991. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, CA.

Appelbe, W.F., and Hansen, K. 1985. A survey of systems programming languages: concepts and facilities. *Software—Practice and Experience 15*, 2 (February), 169–190.

Bal, H.E., Steiner, J.G., and Tanenbaum, A.S. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys 21*, (September), 261–322.

Bauer, F.L., and Samelson, K. 1976. Language hierarchies and interfaces. Proceedings of an International Summer School at Marktoberdorf, Germany, July 23–August 2, 1975. *Lecture Notes in Computer Science 46*, Springer-Verlag, New York.

Bell, D.H., Kerridge, J.M., Simpson, D., and Willis, N. 1983. *Parallel Programming—A Bibliography.* Wiley Heyden, New York.

Bell, J.R. 1973. Threaded code. *Communications of the ACM 16*, 6 (June), 370–372.

Ben-Ari, M. 1982. *Principles of Concurrent Programming.* Prentice-Hall, Englewood Cliffs, NJ.

Bergland, G.D., and Gordon, R.D., Eds. 1981. *Software Design Strategies.* IEEE Computer Society, Los Angeles, CA.

Bernstein, A.J., and Ensor, J.R. 1981. A Modula based language supporting hierarchical development and verification. *Software—Practice and Experience 11*, 3 (March), 237–255.

Bishop, J. 1986. *Data Abstraction in Programming Languages.* Addison-Wesley, Reading, MA.

Black, A., Hutchinson, N., Jul, E., and Levy, H. 1986. Object structure in the Emerald system. *SIGPLAN Notices 21*, (November), 78–86.

Bochmann, G.V., and Joachim, T. 1979. The development and structure of an X.25 implementation. *IEEE Transactions on Software Engineering 5*, 5 (May), 429–439.

Boyle, J., Butler, R., Disz, T., Glickfeld, B., Lusk, E., Overbeek, R., Patterson, J., and Stevens, R. 1987. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, New York.

Brinch Hansen, P. 1969. *RC 4000 Software: Multiprogramming System.* Regnecentralen, Copenhagen, Denmark, (April). Revised version in Brinch Hansen (1973b), 237–286.

Brinch Hansen, P. 1971a. An outline of a course on operating system principles. *Seminar on Operating Systems Techniques*, Belfast, Northern Ireland, (August). In Hoare (1972a), 29–36. Review: *Computing Reviews*, 26738, (1973). *Article 3.*

Brinch Hansen, P. 1971b. Short-term scheduling in multiprogramming systems. *ACM Symposium on Operating Systems Principles*, Palo, Alto, CA, (October), 101–105. Review: Bibliography 27, *Computing Reviews*, (1972).

Brinch Hansen, P. 1971c. Multiprogramming with monitors. Carnegie-Mellon University, Pittsburgh, PA, (November). Privately circulated.

Brinch Hansen, P. 1972a. A comparison of two synchronizing concepts. *Acta Informatica 1*, 190–199. Submitted November 1971. Review: *Computing Reviews*, 26837, (1974).

Brinch Hansen, P. 1972b. Structured multiprogramming. Invited paper, *Communications of the ACM 15*, 7 (July), 574–578. Also in Gries (1978), 215–223. Review: *Computing Reviews*, 24238, (1972). *Article 4.*

Brinch Hansen, P. 1973a. A reply to comments on "A comparison of two synchronizing concepts." *Acta Informatica 2*, 189–190.

Brinch Hansen, P. 1973b. *Operating System Principles*, Prentice-Hall. Englewood Cliffs, NJ, (July). Submitted May 1972. Translations: Kindai Kagaku Sha, Tokyo, Japan, 1976; Carl Hanser Verlag, Munich, Germany, 1977; SNTL, Prague, Czechoslovakia, 1979; Wydawnictwa Naukowo-Techniczne, Warsaw, Poland, 1979; Naučna Knjiga, Belgrade, Yugoslavia, 1982. Reviews: *Computing Reviews*, 26104, (1973), and 29801, (1976); *American Scientist, Computer, BIT*, (1975); *Embedded Systems Programming*, (1990).

Brinch Hansen, P. 1973c. On September 6, 1973, I sent Mike McKeag "a copy of a preliminary document that describes my suggestion for an extension of Pascal with concurrent processes and monitors" (McKeag 1991). No longer available.

Brinch Hansen, P. 1973d. Concurrent programming concepts. Invited paper, *ACM Computing Surveys 5*, 4 (December), 223–245. Review: *Computing Reviews*, 26927, (1974).

Brinch Hansen, P. 1974a. Concurrent Pascal: a programming language for operating system design. Information Science, California Institute of Technology, Pasadena, CA, (April). Referenced in Silberschatz (1977). No longer available.

Brinch Hansen, P. 1974b. Deamy—a structured operating system. Information Science, California Institute of Technology, Pasadena, CA, (May). Referenced in Brinch Hansen (1975c).

Brinch Hansen, P. 1974c. A programming methodology for operating system design. Invited paper, *Proceedings of the IFIP Congress 74*, (August), 394–397, North-Holland, Amsterdam, The Netherlands. Review: *Computing Reviews*, 27985, (1975).

Brinch Hansen, P. 1974d. The programming language Concurrent Pascal, (Part I. The purpose of Concurrent Pascal; Part II. The use of Concurrent Pascal). Information Science, California Institute of Technology, Pasadena, CA, (November). Revised February 1975. Also in Proceedings of the International Conference on Reliable Software, Los Angeles, CA, April 1975, *SIGPLAN Notices 10*, 6 (June), 305–309 (Part I only); Invited paper, *IEEE Transactions on Software Engineering 1*, 2 (June 1975), 199–207; Bauer (1976), 82–110; Gries (1978), 224–261; Wasserman (1980), 465–473; Kuhn (1981), 313–321; Horowitz (1983b), 262–272; Gehani (1988), 73–92. Review: *Computing Reviews*, 29418, (1976). *Article 7.*

Brinch Hansen, P. 1975a. Concurrent Pascal report. Information Science, California Institute of Technology, Pasadena, CA, (June). Also in Brinch Hansen (1977b), 231–270.

Brinch Hansen, P., and Hartmann, A.C. 1975b. Sequential Pascal report. Information Science, California Institute of Technology, Pasadena, CA, (July).

Brinch Hansen, P. 1975c. The Solo operating system. Information Science, California Institute of Technology, Pasadena, CA, (June–July). Also in *Software—Practice and Experience 6*, 2 (April–June 1976), 141–200; Brinch Hansen (1977b), 69–142. Review: *Computing Reviews*, 31363, (1977). *Articles 8–9.*

Brinch Hansen, P. 1975d. Universal types in Concurrent Pascal. *Information Processing Letters 3*, (July), 165–166.

Brinch Hansen, P. 1975e. Concurrent Pascal machine. Information Science, California Institute of Technology, Pasadena, CA, (October). Also in Brinch Hansen (1977b), 271–297.

Brinch Hansen, P. 1975f. A real-time scheduler. Information Science, California Institute of Technology, Pasadena, CA, (November). Also in Brinch Hansen (1977b), 189–227.

Brinch Hansen, P. 1976a. The job stream system. Information Science, California Institute of Technology, Pasadena, CA, (January). Also in Brinch Hansen (1977b), 148–188.

Brinch Hansen, P. 1976b. Concurrent Pascal implementation notes. Information Science, California Institute of Technology, Pasadena, CA. Referenced in Powell (1979). No longer available.

Brinch Hansen, P. 1977a. Experience with modular concurrent programming. *IEEE Transactions on Software Engineering 3*, 2 (March), 156–159. *Article 11.*

Brinch Hansen, P. 1977b. *The Architecture of Concurrent Programs.* Prentice-Hall, Englewood Cliffs, NJ, (July). Submitted July 1976. Translations: Kagaku-Gijyutsu, Tokyo, Japan, 1980; Oldenbourg, Munich, Germany, 1981. Reviews: *Choice, Ingeniøren*, (1978); *Computing Reviews*, 33358, *Computer*, (1979); *Embedded Systems Programming*, (1990).

Brinch Hansen, P. 1977c. Network—a multiprocessor program. *IEEE Computer and Software Applications Conference*, Chicago, IL, (November), 336–340. Also in *IEEE Transactions on Software Engineering 4*, 3 (May 1978), 194–199. Review: *Computing Reviews*, 33840, (1978). *Article 13.*

Brinch Hansen, P., and Staunstrup, J. 1978a. Specification and implementation of mutual exclusion. *IEEE Transactions on Software Engineering 4*, 5 (September), 365–370.

Brinch Hansen, P. 1978b. Distributed Processes: a concurrent programming concept. *Communications of the ACM 21*, 11 (November), 934–941. Submitted September 1977, revised December 1977. Also in Bergland (1981), 289–296; Saib (1983), 500–507; Gehani (1988), 216–233. *Article 14.*

Brinch Hansen, P. 1978c. A keynote address on concurrent programming. *IEEE Computer Software and Applications Conference*, Chicago, IL, (November), 1–6. Also in *Computer 12*, (May 1979), 50–56; *Selected Reprints in Software*, M.V. Zelkowitz, Ed., IEEE Computer Society, Los Angeles, CA, (1982), 42–48. Review: *Computing Reviews*, 35247, (1979). *Article 16.*

Brinch Hansen, P. 1978d. Reproducible testing of monitors. *Software—Practice and Experience 8*, 6 (November–December), 721–729. *Article 15.*

Brinch Hansen, P., and Fellows, J.A. 1980. The Trio operating system. Computer Science Department, University of Southern California, Los Angeles, CA, (June). Also in *Software—Practice and Experience 10*, 11 (November 1980), 943–948. Review: *Computing Reviews*, 37637, (1981).

Brinch Hansen, P. 1981. The design of Edison. *Software—Practice and Experience 11*, 4 (April), 363–396. *Article 17.*

Brinch Hansen, P. 1989a. The Joyce language report. *Software—Practice and Experience 19*, 6 (June), 553–578.

Brinch Hansen, P. 1989b. A multiprocessor implementation of Joyce. *Software—Practice and Experience 19*, 6 (June), 579–592. *Article 19.*

Bryant, R.E., and Dennis, J.B. 1979. Concurrent programming. In Wegner (1979), 584–610.

Burns, A., and Davies, G. 1988. Pascal-FC: a language for teaching concurrent programming. *SIGPLAN Notices 23*, (January), 58–66.

Bustard, D.W., Elder, J., and Welsh, J. 1988. *Concurrent Program Structures*. Prentice-Hall, Englewood Cliffs, NJ.

Coleman, D., Gallimore, R.M., Hughes, J.W., and Powell, M.S. 1979. An assessment of Concurrent Pascal. *Software—Practice and Experience 9*, 10 (October), 827–837. Also in Gehani (1988), 351–364.

Coleman, D. 1980. Concurrent Pascal—an appraisal. In McKeag (1980), 213–227.

Courtois, P.J., Heymans, F., and Parnas, D.L. 1971. Concurrent control with "readers" and "writers." *Communications of the ACM 14*, 10 (October), 667–668.

Courtois, P.J., Heymans, F., and Parnas, D.L. 1972. Comments on "A comparison of two synchronizing concepts." *Acta Informatica 1*, 375–376.

Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. 1972a. *Structured Programming*. Academic Press, New York.

Dahl, O.-J., and Hoare, C.A.R. 1972b. Hierarchical program structures. In Dahl (1972a), 175–220.

Deitel, H.M. 1984. *An Introduction to Operating Systems*. Revised first edition, Addison-Wesley, Reading, MA.

Dijkstra, E.W. 1965. Cooperating sequential processes. Mathematical Department, Technological University, Eindhoven, The Netherlands, (September). Also in Genuys (1968), 43–112.

Dijkstra, E.W. 1967. The structure of the "THE"-multiprogramming system. *ACM Symposium on Operating System Principles*, Gatlinburg, TN. Also in *Communications of the ACM 11*, 5 (May 1968), 341–346, and *26*, (January 1983), 49–52.

Dijkstra, E.W. 1971. Hierarchical ordering of sequential processes. *Acta Informatica 1*, 115–138. Also in Hoare (1972a), 72–93.

Discussion. 1971. Discussion of conditional critical regions and monitors. *Seminar on Operating Systems Techniques*, Belfast, Northern Ireland, (August). In Hoare (1972a), 110–113.

Dunman, B.R., Schack, S.R., and Wood, P.T. 1982. A mainframe implementation of Concurrent Pascal. *Software—Practice and Experience 12*, 1 (January), 85–90.

Fellows, J.A. 1980. Applications of abstract data types: The Trio operating system. Ph.D. thesis, Computer Science Department, University of Southern California, Los Angeles, CA.

Gehani, N., and McGettrick, A.D., Eds. 1988. *Concurrent Programming*. Addison-Wesley, Reading, MA.

Genuys, F., Ed. 1968. *Programming Languages*. Academic Press, New York.

Ghezzi, C., and Jazayeri, M. 1982. *Programming Language Concepts*. John Wiley, New York.

Graef, N., Kretschmer, H., Löhr, K.-P., and Morawetz, B. 1979. How to design and implement small time-sharing systems using Concurrent Pascal. *Software—Practice and Experience 9*, 1 (January), 17–24.

Gries, D., Ed. 1978. *Programming Methodology—A Collection of Articles by Members of IFIP WG2.3*. Springer-Verlag, New York.

Haddon, B.K. 1977. Nested monitor calls. *Operating Systems Review 11*, (October), 18–23.

Hartmann, A.C. 1975. A Concurrent Pascal compiler for minicomputers. Ph.D. thesis, Information Science, California Institute of Technology, Pasadena, CA, (September). Also published as *Lecture Notes in Computer Science 50*, (1977), Springer-Verlag, New York.

Hayden, C. 1979. Distributed processes: experience and architectures. Ph.D. thesis, Computer Science Department, University of Southern California, Los Angeles, CA.

Heimbigner, D. 1978. Writing device drivers in Concurrent Pascal. *Operating Systems Review 12*, 4 (April), 16–33.

Hoare, C.A.R. 1971a. Towards a theory of parallel programming. Queen's University, Belfast, Northern Ireland, (August). Privately circulated. Not to be confused with Hoare (1971b) of the same title.

Hoare, C.A.R. 1971b. Towards a theory of parallel programming. *Seminar on Operating Systems Techniques*, Belfast, Northern Ireland, (August). In Hoare (1972a), 61–71. Also in Gries (1978), 202–214. Not to be confused with Hoare (1971a) of the same title.

Hoare, C.A.R., and Perrott, R.H., Eds. 1972a. *Operating Systems Techniques*, Proceedings of a seminar at Queen's University, Belfast, August 30–September 3, 1971. Academic Press, New York.

Hoare, C.A.R. 1972b. Proof of correctness of data representations. *Acta Informatica 1*, 271–281. Submitted February 1972. Also in Bauer (1976), 183–193; Gries (1978), 269–281; Hoare (1989), 103–115.

Hoare, C.A.R. 1973a. A pair of synchronising primitives. On January 11, 1973, Hoare gave Jim Horning a copy of this undated, unpublished draft (Horning 1991).

Hoare, C.A.R. 1973b. A structured paging system. *Computer Journal 16*, (August), 209–214. Submitted October 1972. Also in Hoare (1989), 133–151.

Hoare, C.A.R. 1974a. Monitors: an operating system structuring concept. *Communications of the ACM 17*, (October), 549–557. Submitted February 1973, revised April 1974. Also in Gries (1978), 224–243; Wasserman (1980), 156–164; Gehani (1988), 256–277; Hoare (1989), 171–191.

Hoare, C.A.R. 1974b. Hints on programming language design. In *Computer Systems Reliability*, C. Bunyan, Ed., Infotech International, Berkshire, England, 505–534. Also in Wasserman (1980), 43–52; Hoare (1989), 193–216.

Hoare, C.A.R. 1976. Hints on the design of a programming language for real-time command and control. In *Real-time Software: International State of the Art Report*, J.P. Spencer, Ed., Infotech International, Berkshire, England, 685–699.

Hoare, C.A.R. 1978. Communicating sequential processes. *Communications of the ACM 21*, (August), 666–677. Submitted March 1977, revised August 1977. Also in Wasserman (1980), 170–181; Bergland (1981), 277–288; Kuhn (1981), 323–334; *Communications of the ACM 26*, (January 1983), 100–106; Horowitz (1983b), 306–317; Saib (1983), 508–519; Gehani (1988), 278–308; Hoare (1989), 259–288.

Hoare, C.A.R., and Jones, C.B., Ed., 1989. *Essays in Computing Science*. Prentice-Hall, Englewood Cliffs, NJ.

Holt, R.C., Graham, G.S., Lazowska, E.D., and Scott, M.A. 1978. *Structured Concurrent Programming with Operating Systems Applications*. Addison-Wesley, Reading, MA.

Holt, R.C. 1982. A short introduction to Concurrent Euclid. *SIGPLAN Notices 17*, (May), 60–79.

Holt, R.C. 1983. *Concurrent Euclid, the Unix System and Tunis.* Addison-Wesley, Reading, MA.

Holt, R.C. 1988. Device management in Turing Plus. *Operating System Review 22*, 1 (January), 33–41.

Horning, J.J. 1991. Personal communication, (May).

Horowitz, E. 1983a. *Fundamentals of Programming Languages.* Computer Science Press, Rockville, MD.

Horowitz, E., Ed. 1983b. *Programming Languages: A Grand Tour.* Computer Science Press, Rockville, MD.

Howard, J.H. 1976a. Proving monitors. *Communications of the ACM 19*, 5 (May), 273–279.

Howard, J.H. 1976b. Signalling in monitors. *IEEE Conference on Software Engineering*, San Francisco, CA, (October), 47–52.

Janson, P.A. 1985. *Operating Systems: Structures and Mechanisms.* Academic Press, New York.

Kaubisch, W.H., Perrott, R.H., and Hoare, C.A.R. 1976. Quasiparallel programming. *Software—Practice and Experience 6*, (July–September), 341–356.

Keedy, J.L. 1978. On structuring operating systems with monitors. *Australian Computer Journal 10*, 1 (February), 23–27.

Kerridge, J.M. 1982. A Fortran implementation of Concurrent Pascal. *Software—Practice and Experience 12*, 1 (January), 45–56.

Kessels, J.L.W. 1977. An alternative to event queues for synchronization in monitors. *Communications of the ACM 20*, 7 (July), 500–503.

Kligerman, E., and Stoyenko, A.D. 1986. Real time Euclid: a language for reliable real time systems. *IEEE Transactions on Software Engineering 12*, 9 (September), 941–949.

Kotulski, L. 1987. About the semantic nested monitor calls. *SIGPLAN Notices 22*, 4 (April), 80–82.

Krakowiak, S. 1988. *Principles of Operating Systems.* MIT Press, Cambridge, MA.

Krishnamurthy, E.V. 1989. *Parallel Processing: Principles and Practice.* Addison-Wesley, Reading, MA.

Kruijer, H.S.M. 1982a. Processor management in a Concurrent Pascal kernel. *Operating Systems Review 16*, (April), 7–17.

Kruijer, H.S.M. 1982b. A multi-user operating system for transaction processing written in Concurrent Pascal. *Software—Practice and Experience 12*, 5 (May), 445–454.

Kuhn, R.H., and Padua, D.A., Eds. 1981. *Parallel Processing.* IEEE Computer Society, Los Angeles, CA, (August).

Lampson, B.W., and Redell, D.D. 1980. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (February), 105–117. Also in Gehani (1988), 392–418.

Lauer, H.C., and Needham, R.M. 1978. On the duality of operating system structures. *International Symposium on Operating Systems*, IRIA, France (October). Also in *Operating Systems Review 13*, 2 (April 1979), 3–19.

Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J.C., Scheifler, R., and Snyder, A. 1981. CLU reference manual. *Lecture Notes in Computer Science 114*. Quote: “By the summer of 1975, the first version of the language had been completed. Over the next two years, the entire language design was reviewed and two implementations were produced ... A preliminary version of this manual appeared in July 1978” (p. III).

Lister, A.M. 1977. The problem of nested monitor calls. *Operating Systems Review 11*, (July), 5–7.

Löhr, K.-P. 1977. Beyond Concurrent Pascal. *SIGPLAN Notices 12*, (November), 128–137.

Lynch, W.C. 1991. Personal communication, (October).

McKeag, R.M. 1973. Programming languages for operating systems. *ACM SIGPLAN/SIGOPS Interface Meeting*, Savannah, GA, (April). In *SIGPLAN Notices 8*, 9 (September 1973), 109–111.

McKeag, R.M., and Macnaghten, A.M., Eds. 1980. *On the Construction of Programs.* Cambridge University Press, New York.

McKeag, R.M. 1991. Personal communication, (August).

Maddux, R.A., and Mills, H.D. 1979. Review of The Architecture of Concurrent Programs. *IEEE Computer 12*, (May), 102–103.

Mattson, S.E. 1980. Implementation of Concurrent Pascal on LSI-11. *Software—Practice and Experience 10*, 3 (March), 205–218.

Møller-Nielsen, P., and Staunstrup, J. 1984. Experiments with a multiprocessor. Computer Science Department, Aarhus University, Aarhus, Denmark (November).

Narayana, K.T., Prasad, V.R., and Joseph, M. 1979. Some aspects of concurrent programming in CCNPascal. *Software—Practice and Experience 9*, 9 (September), 749–770.

Neal, D., and Wallentine, V. 1978. Experiences with the portability of Concurrent Pascal. *Software—Practice and Experience 8*, 3 (May–June), 341–354.

Nehmer, J. 1979. The implementation of concurrency for a PL/I-like language. *Software—Practice and Experience 9*, 12 (December), 1043–1057.

Nori, K.V., Ammann, U., Jensen, K., and Naegeli, H.H. 1974. The Pascal P compiler: implementation notes. Institut für Informatik, ETH, Zurich, Switzerland, (December).

Nutt, G.J. 1992. *Centralized and Distributed Operating Systems.* Prentice-Hall, Englewood Cliffs, NJ.

Parnas, D.L. 1978. The non-problem of nested monitor calls. *Operating Systems Review 12*, (January), 12–14.

Perrott, R.H. 1987. *Parallel Programming.* Addison-Wesley, Reading, MA.

Peterson, J.L., and Silberschatz, A. 1983. *Operating Systems Concepts.* Addison-Wesley, Reading, MA.

Pinkert, J.R., and Wear, L.L. 1989. *Operating Systems: Concepts, Policies and Mechanisms.* Prentice-Hall, Englewood Cliffs, NJ.

Popek, G.J., Horning, J.J., Lampson, B.W., Mitchell, J.G., and London, R.L. 1977. Notes on the design of Euclid. *SIGPLAN Notices 12*, 11–12. Quote: “The System Development Corporation is currently implementing Euclid” (p. 12).

Powell, M.S. 1979. Experience of transporting and using the Solo operating system. *Software—Practice and Experience 9*, 7 (July), 561–570.

Ravn, A.P. 1982. Use of Concurrent Pascal in systems programming teaching. *Microprocessing and Microprogramming 10*, 33–35.

Raynal, M., and Helary, J.-M. 1990. *Synchronization and Control of Distributed Systems and Programs.* John Wiley, New York.

Ringstöm, J. 1990. Predula: a multi-paradigm parallel programming environment. Department of Computer and Information Science, Linköping University, Linköping, Sweden, (November).

Roubine, O., and Heliard, J.-C. 1980. Parallel processing in Ada. In McKeag (1980), 193–212. Also in Gehani (1988), 142–159.

Saib, S.H., and Fritz, R.E., Eds. 1983. *The Ada Programming Language.* IEEE Computer Society, Los Angeles, CA.

Schneider, H.J. 1984. *Problem Oriented Programming Languages.* John Wiley, New York.

Sebesta, R.W. 1989. *Concepts of Programming Languages.* Benjamin/Cummings, Redwood City, CA.

Shaw, M., Ed. 1981. *Alphard: Form and Content.* Springer-Verlag, New York. Quotes: "The preliminary language report appeared as a . . . technical report [in February 1978]. No final report was issued;" . . . "We curtailed development of the compiler in 1979 when it became clear that another iteration on the language design was necessary" (pp. 191 and 315).

Silberschatz, A., Kieburtz, R.B., and Bernstein, A.J. 1977. Extending Concurrent Pascal to allow dynamic resource management. *IEEE Transactions on Software Engineering 3*, (May), 210–217.

Staunstrup, J. 1978. Specification, verification, and implementation of concurrent programs. Ph.D. thesis, Computer Science Department, University of Southern California, Los Angeles, CA, (May).

Stotts, P.D. 1982. A comparative survey of concurrent programming languages. *SIGPLAN Notices 17*, 10 (October), 76–87. Also in Gehani (1988), 419–435.

Tanenbaum, A.S. 1992. *Modern Operating Systems.* Prentice-Hall, Englewood Cliffs, NJ.

Tennent, R.D. 1981. *Principles of Programming Languages.* Prentice-Hall, Englewood Cliffs, NJ.

Tsichritzis, D.C., and Bernstein, P.A. 1974. *Operating Systems.* Academic Press, New York.

Tsujino, Y., Ando, M., Araki, T., and Tohura, N. 1984. Concurrent C: a programming language for distributed multiprocessor systems. *Software—Practice and Experience 14*, 11 (November), 1061–1078.

Turski, W.M. 1978. *Computer Programming Methodology.* Heyden, Philadelphia, PA.

Wasserman, A.I., Ed. 1980. *Programming Language Design.* IEEE Computer Society, Los Angeles, CA, (October).

Wegner, P., Ed. 1979. *Research Directions in Software Technology.* MIT Press, Cambridge, MA.

Welsh, J., and Bustard, D.W. 1979. Pascal-Plus—another language for modular multiprogramming. *Software—Practice and Experience 9*, 11 (November), 947–957.

Welsh, J., and McKeag, R.M. 1980. *Structured System Programming.* Prentice-Hall, Englewood Cliffs, NJ.

Wettstein, H. 1978. The problem of nested monitor calls revisited. *Operating Systems Review 12*, (January), 19–23.

Whiddett, R.J. 1983. Dynamic distributed systems. *Software—Practice and Experience 13*, 4 (April), 355–371.

Whiddett, R.J. 1987. *Concurrent Programming for Software Engineers.* Halstead Press, New York.

White, J.E. 1976. A high-level framework for network-based resource sharing. *National Computer Conference*, AFIPS Press, Montvale, NJ, (June), 561–570.

Williams, S.A. 1990. *Programming Models for Parallel Systems.* John Wiley, New York.

Wilson, L.B., and Clark, R.G. 1988. *Comparative Programming Languages.* Addison-Wesley, Reading, MA.

Wirth, N. 1971. The programming language Pascal. *Acta Informatica 1*, 35–63.

Wirth, N. 1977. Modula: a programming language for modular multiprogramming. *Software—Practice and Experience 7*, 1 (January–February), 3–35. Also in Horowitz (1983b), 273–305.

Young, S.J. 1982. *Real Time Languages: Design and Development.* Halstead Press, New York.