THE INVENTION OF CONCURRENT PROGRAMMING

PER BRINCH HANSEN

(2001)

The author selects classic papers written by the computer scientists who made the major breakthroughs in concurrent programming. These papers cover the pioneering era of the field from the semaphores of the mid 1960s to the remote procedure calls of the late 1970s. The author summarizes the classic papers and puts them in historical perspective.

A PROGRAMMING REVOLUTION

This is the story of one of the major revolutions in computer programming: *the invention of concurrent programming*.

Tom Kilburn and David Howarth pioneered the use of interrupts to simulate concurrent execution of several programs on the *Atlas computer* (Kilburn 1961). This programming technique became known as *multiprogramming*.

The early multiprogramming systems were programmed in assembly language without any conceptual foundation. The slightest programming mistake could make these systems behave in a completely erratic manner that made program testing nearly impossible.

By the end of the 1960s multiprogrammed operating systems had become so huge and unreliable that their designers spoke openly of a *software crisis* (Naur 1969).

As J. M. Havender (1968) recalled:

> The original multitasking concept of the [*IBM OS/360*] envisioned relatively unrestrained competion for resources to perform a number of tasks concurrently ... But as the system evolved many instances of task deadlock were uncovered.

Elliott Organick (1973) pointed out that the termination of a task in the *Burroughs B6700* system might cause its offspring tasks to lose their stack space!

In the mid 1960s computer scientists took the first steps towards a deeper understanding of concurrent programming. In less than fifteen years, they discovered *fundamental concepts*, expressed them by *programming notation*, included them in *programming languages*, and used these languages to write *model operating systems*. In the 1970s the new programming concepts were used to write the first *concise textbooks* on the principles of *operating systems* and *concurrent programming*.

The development of concurrent programming was originally motivated by the desire to develop reliable operating systems. From the beginning, however, it was recognized that principles of concurrent programming "have a general utility that goes beyond operating systems"—they apply to *any* form of parallel computing (Brinch Hansen 1971a).

I would like to share the excitement of these discoveries with you by offering my own assessment of the classic papers in concurrent programming.

This essay is not just an editorial overview of the selected papers. It is also my personal reflections on the major contributions, which inspired me (and others) in our common search for simplicity in concurrent programming.

If you compare my early papers with this essay, you will notice an occasional change of perspective. With thirty years of hindsight, that is inevitable.

I have made an honest attempt to rely only on the *publication record* to document historic events and settle issues of priority. However, as a contributor to these ideas I cannot claim to have written an unbiased account of these events. That can only be done by a professional historian.

THE CLASSIC PAPERS

Choosing the *classic papers* was easier than I thought:

- *First I made a short list of fundamental contributions to abstract concurrent programming of major and lasting technical importance.*

- *Then I selected the original papers in which computer scientists first introduced these ideas.*

- *I added a few papers that illustrate the influence of concurrent programming concepts on operating system principles and programming language implementation.*

- *Finally I put the papers in chronological order to illustrate how each new idea was motivated by earlier successes and failures.*

Fundamental Concepts

Asynchronous processes
Speed independence
Fair scheduling
Mutual exclusion
Deadlock prevention
Process communication
Hierarchical structure
Extensible system kernels

Programming Language Concepts

Concurrent statements
Critical regions
Semaphores
Message buffers
Conditional critical regions
Secure queueing variables
Monitors
Synchronous message communication
Remote procedure calls

Classic Papers

1. *E. W. Dijkstra, Cooperating Sequential Processes (1965).*
2. *E. W. Dijkstra, The Structure of the THE Multiprogramming System (1968).*
3. *P. Brinch Hansen, RC 4000 Software: Multiprogramming System (1969).*
4. *E. W. Dijkstra, Hierarchical Ordering of Sequential Processes (1971).*
5. *C. A. R. Hoare, Towards a Theory of Parallel Programming (1971).*
6. *P. Brinch Hansen, An Outline of a Course on Operating System Principles (1971).*
7. *P. Brinch Hansen, Structured Multiprogramming (1972).*
8. *P. Brinch Hansen, Shared Classes (1973).*
9. *C. A. R. Hoare, Monitors: An Operating System Structuring Concept (1974).*
10. *P. Brinch Hansen, The Programming Language Concurrent Pascal (1975).*
11. *P. Brinch Hansen, The Solo Operating System: A Concurrent Pascal Program (1976).*
12. *P. Brinch Hansen, The Solo Operating System: Processes, Monitors and Classes (1976).*
13. *P. Brinch Hansen, Design Principles (1977).*
14. *E. W. Dijkstra, A Synthesis Emerging? (1975).*
15. *C. A. R. Hoare, Communicating Sequential Processes (1978).*
16. *P. Brinch Hansen, Distributed Processes: A Concurrent Programming Concept (1978).*
17. *P. Brinch Hansen, Joyce—A Programming Language for Distributed Systems (1987).*
18. *P. Brinch Hansen, SuperPascal: A Publication Language for Parallel Scientific Computing (1994).*
19. *P. Brinch Hansen, Efficient Parallel Recursion (1995).*

After following this selection procedure rigorously, I was surprised to see that *every* single paper turned out to have been written by either Edsger Dijkstra, Tony Hoare or me. In retrospect, this was, perhaps, not so surprising.

In Judy Bishop's (1986) view:

> The swing away from assembly language which gained genuine momentum during the seventies was slow to affect the area of concurrent systems—operating systems, embedded control systems and the like. What happened was that three people—Edsger Dijkstra, Tony

> Hoare and Per Brinch Hansen—independently developed key abstractions which were taken up by researchers worldwide, realized in experimental languages, reported on, adapted and refined. In this way, the problems of concurrency could be expressed in well understood notation, and solutions and principles gradually evolved.

To produce an anthology of reasonable size (about 500 pages) *I omitted*:

- Subsequent work that built on the seminal papers without adding anything fundamentally new.
- Survey papers and assessments of ideas.
- Implementation details (except in outline).
- Testing, verification, and formal theory.
- Functional multiprogramming and data parallel languages.

These guidelines eliminated many valuable contributions to concurrent programming (as well as two dozen of my own papers). Some of them are listed as recommended *further reading* in the *bibliography* at the end of this essay.

PART I CONCEPTUAL INNOVATION

It is difficult for students today to imagine how little anyone knew about systematic programming in the early 1960s. Let me illustrate this by telling you about my first modest experience with multiprogramming.

In 1963 I graduated from the Technical University of Denmark without any programming experience (it was not yet being taught). There were (as far as I remember) no textbooks available on programming languages, compilers or operating systems.

After graduating I joined the Danish computer company Regnecentralen. Working on a Cobol compiler project, headed by Peter Naur and Jørn Jensen, I taught myself to program.

In 1966 Peter Kraft and I were asked to design a real-time system for supervising a large ammonia nitrate plant in Poland. A small computer would be used to perform a fixed number of cyclical tasks simultaneously. These tasks would share data tables and peripherals. Since plant operators could change the frequencies of individual tasks (and stop some of them indefinitely), we could not make any assumptions about the relative (or absolute) speeds of the tasks.

It was obvious that we needed multiprogramming with process synchronization. But what kind of synchronization?

A common technique at the time was to *suspend* a process in a queue until it was *resumed* by another process. The trouble was that resumption had no effect if the queue was empty. This happened if resumption was attempted *before* a process was suspended. (This pitfall reminds me of a mailman who throws away your letters if you are not at home when he attempts to deliver them!)

This mechanism is unreliable because it makes a seemingly innocent assumption about the relative timing of parallel events: A process must *never* attempt to resume another process that is not suspended. However, since the timing of events is unpredictable in a real-time system, this would have been a disastrous choice for our real-time system.[1]

Regnecentralen had no experience with multiprogramming. Fortunately, Edsger Dijkstra was kind enough to send me a copy of his 1965 monograph "Cooperating Sequential Processes," with a personal dedication: "Especially made for graceful reading!" (I still have it.)

Using Dijkstra's semaphores, Peter Kraft, Charles Simonyi and I were able to implement the *RC 4000 real-time control system* on the prototype of Regnecentralen's RC 4000 computer with only 4K words of memory (without a drum or disk) (Brinch Hansen 1967a, 1967b).

1 Cooperating Sequential Processes

The first classic is one of the great works in computer programming:

E. W. Dijkstra, Cooperating Sequential Processes (1965)

Here Dijkstra lays the *conceptual foundation* for *abstract concurrent programming*. He begins by making the crucial assumption about *speed independence*:

> We have stipulated that processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as

[1]Around 1965 IBM's *PL/I* language included queueing variables of this kind known as *events*. Surprisingly, the *suspend* and *resume* primitives are also included in the recent *Java* language (Doug Lea 1997).

completely independent of each other. In particular, we disallow any assumption about the relative speeds of the different processes.

Indivisible operations were well-known in multiprogramming systems, in the form of *supervisor calls* (Kilburn 1961). Dijkstra's contribution was to make explicit assumptions about these *critical sections* (as he calls them).[2]

For pedagogical reasons, Dijkstra first attempts to program critical sections using assignments and inspection of simple variables only.

Through a carefully presented sequence of rejected solutions, Dijkstra arrives at the following *correctness criteria* for cyclical processes cooperating by means of common variables and critical sections:

1. *Mutual exclusion*: "At any moment at most one of the processes is engaged in its critical section."
2. *Fairness*: "The decision which of the processes is the first to enter its critical section cannot be postponed to eternity."
3. *Speed independence*: "Stopping a process in its 'remainder of cycle' [that is, outside its critical region] has no effect upon the others."

The Dutch mathematician T. J. Dekker found a general solution to the mutual exclusion problem without synchronizing primitives. For single-processor systems, I have always viewed this as an ingenious, academic exercise. Computer designers had solved the problem (in a restricted way) by the simple technique of *disabling interrupts*.

As a more realistic solution, Dijkstra introduces *binary semaphores*, which make the mutual exclusion problem trivial.[3]

Using *general semaphores* (due to Carel Scholten), Dijkstra implements message communication through a *bounded buffer*.[4] He achieves a pleasing *symmetric behavior* of communicating processes by viewing senders as processes that consume empty buffer slots and produce full slots. Similarly, receivers consume full slots and produce empty ones.

Dijkstra also presents an ingenious method of *deadlock prevention*, known as *the banker's algorithm*.

[2]Hoare (1971) renamed them *critical regions*.

[3]Dijkstra used Dutch *acronyms*, *P* and *V*, for the semaphore operations. Being allergic to acronyms in any language, I renamed them *wait* and *signal* (Brinch Hansen 1971a).

[4]The *bounded buffer* is used as a programming example throughout this essay.

```
begin integer number of queuing portions,
              number of empty positions,
              buffer manipulation;
      number of queuing portions:= 0;
      number of empty positions:= N;
      buffer manipulation:= 1;
      parbegin
      producer: begin
             again 1: produce next portion;
                      P(number of empty positions);
                      P(buffer manipulation);
                      add portion to buffer;
                      V(buffer manipulation);
                      V(number of queuing portions);
                      goto again 1
                end;
      consumer: begin
             again 2: P(number of queuing portions);
                      P(buffer manipulation);
                      take portion from buffer;
                      V(buffer manipulation);
                      V(number of empty positions);
                      process portion taken;
                      goto again 2
                end
      parend
end
```

The Bounded Buffer with Semaphores

In the 1960s Alan Perlis noticed that Regnecentralen's compiler group discussed programming problems by writing *Algol 60* statements on a blackboard. This was unusual at a time when systems programs were still being written in assembly language.

Edsger Dijkstra was also firmly in the Algol 60 tradition (Naur 1960). He writes parallel algorithms in Algol extended with a *parallel statement*:[5]

parbegin S_1; S_2; ... S_n **parend**

As Dijkstra defines it:

Initiation of a parallel compound implies simultaneous initiation of all

[5]Also known as a *concurrent statement* (Brinch Hansen 1972b).

> its constituent statements, its execution is completed after the completion of the execution of all its constituent statements.

This modest proposal is one of the first published examples of an *abstract programming notation* for concurrent processes.

2 THE Multiprogramming System

Dijkstra demonstrated the depth of his ideas in the construction of an elegant *model operating system*:

E. W. Dijkstra, The Structure of the THE Multiprogramming System (1968)

This was a spooling system that compiled and executed a stream of Algol 60 programs with paper tape input and printer output. It used software-implemented demand paging between a 512K word drum and a 32K word memory. There were five user processes and ten input/output processes, one for each peripheral device. The system used *semaphores* for process synchronization and communication.

Dijkstra's multiprogramming system illustrated the conceptual clarity of *hierarchical ordering*. His system consisted of several *program layers*, which gradually transform the physical machine into a more pleasant *abstract machine*:

Level 0: Processor allocation.
Level 1: Demand paging ("segment controller").
Level 2: Operator console ("message interpreter").
Level 3: Virtual devices ("input/output streams").
Level 4: User processes.
Level 5: System operator.

Apart from the operator, these program layers could be designed and tested one at a time.

This short paper concentrates on Dijkstra's most startling claim:

> We have found that it is possible to design a refined multiprogramming system in such a way that its logical soundness can be proved a priori and its implementation can admit exhaustive testing. The only errors that showed up during testing were trivial coding errors . . . the resulting system is guaranteed to be flawless.

The hierarchical structure was used to prove the following properties of *harmoniously cooperating processes*:

1. "Although a process performing a task may in so doing generate a finite number of tasks for other processes, a single initial task cannot give rise to an infinite number of task generations."
2. "It is impossible that all processes have returned to their homing position while somewhere in the system there is still pending a generated but unaccepted task."
3. "After the acceptance of an initial task all processes eventually will be (again) in their homing position."

Software managers continue to believe that software design is based on a magical discipline, called "software engineering," which can be mastered by average programmers. Dijkstra explained that the truth of the matter is simply that

> the intellectual level needed for system design is in general grossly underestimated. I am convinced more than ever that this type of work is very difficult, and that every effort to do it with other than the best people is doomed to either failure or moderate success at enormous expense.

Nico Habermann (1967), Edsger Dijkstra (1971), Coen Bron (1972) and Mike McKeag (1976) described the THE system in more detail.

3 RC 4000 Multiprogramming System

In 1974 Alan Shaw wrote:

> There exist many approaches to multiprogramming system design, but we are aware of only two that are *systematic* and *manageable* and at the same time have been *validated* by producing real working operating systems. These are the hierarchical abstract machine approach developed by Dijkstra (1968a) and the nucleus methods of Brinch Hansen (1969) . . . The nucleus and basic multiprogramming system for the RC 4000 is one of the most elegant existing systems.

The *RC 4000 multiprogramming system* was not a complete operating system, but *a small kernel upon which operating systems for different purposes could be built* in an orderly manner:

P. Brinch Hansen, RC 4000 Software: Multiprogramming System (1969)

The kernel provided the basic mechanisms for creating a *tree of parallel processes* that communicated by messages. Jørn Jensen, Søren Lauesen and I designed it for Regnecentralen's *RC 4000 computer*. We started working on the system in the fall of 1967. A well-documented reliable version was running in the spring of 1969.

Before the RC 4000 multiprogramming system was programmed, I described a *design philosophy* that drastically generalized the concept of an operating system (Brinch Hansen 1968):

> The system has no built-in assumptions about program scheduling and resource allocation; it allows any program to initiate other programs in a hierarchal manner.[6] Thus, the system provides a general frame[work] for different scheduling strategies, such as batch processing, multiple console conversation, real-time scheduling, etc.

This radical idea was probably the most important contribution of the RC 4000 system to operating system technology. If the kernel concept seems obvious today, it is only because it has passed into the general stock of knowledge about system design. It is now commonly referred to as the principle of *separation of mechanism and policy* (Wulf 1974).

The RC 4000 system was also noteworthy for its *message communication. Every communication consisted of an exchange of a message and an answer between two processes.* This protocol was inspired by an early decision to treat peripheral devices as processes, which receive input/output commands as messages and return acknowledgements as answers. In distributed systems this form of communication is now known as *remote procedure calls.*

The system also enabled a server process to be engaged in *nondeterministic communication* with several client processes at a time. This was known as a *conversational process.*

The RC 4000 system was programmed in assembly language. As a purely academic exercise for this essay, I have used an informal Pascal notation (Wirth 1971) to outline a conversational process that implements a *bounded buffer* used by client processes. In retrospect, such a process is equivalent to the "secretary" concept that Dijkstra (1971) would sketch two years later (in very preliminary form).

In the RC 4000 system, the *initial process* was a conversational process that spawned other processes in response to messages from console processes.

[6]Here I obviously meant "processes" rather than "programs."

```
{ The buffer process receives messages from client processes
  requesting it to accept or return data items. The messages
  arrive in buffer elements, which are linked to a message
  queue. The buffer process receives a message in a buffer
  element and uses the same buffer element to return an
  answer to the client process. }
number of items := 0;
{ Inspect the message queue from the beginning }
current buffer := nil;
cycle
  { Postpone receipt of the current buffer element (if any) }
  previous buffer := current buffer;
  { Wait for the next buffer element in the queue (which
    may already have arrived) }
  wait event(previous buffer, current buffer);
  case current buffer.request of
    accept item:
      if number of items < N then
        begin
          take a data item from the current buffer element
            and store it within the buffer process;
          number of items := number of items + 1;
          { Remove the current buffer element from the queue }
          get event(current buffer);
          { Use the same buffer element to return an
            acknowledgment to the client process }
          send answer(acknowledgment, current buffer);
          { Reinspect the queue from the beginning }
          current buffer := nil;
        end;
    return item:
      if number of items > 0 then
        begin
          select a data item stored within the buffer process;
          number of items := number of items - 1;
          { Remove the current buffer element from the queue }
          get event(current buffer);
          { Use the same buffer element to return the
            data item to the client process }
          send answer(data item, current buffer);
          { Reinspect the queue from the beginning }
          current buffer := nil;
        end
   end
end
```

The Bounded Buffer as a Conversational Process

If this *basic operating system* temporarily was unable to honor a request, it would postpone the action by delaying its receipt of the message. In the meantime, it would attempt to serve other clients.

According to Søren Lauesen (1975):

> The RC 4000 software was extremely reliable. In a university environment, the system typically ran under the simple operating system for three months without crashes . . . The crashes present were possibly due to transient hardware errors.

When the RC 4000 system was finished I described it in a 5-page journal paper (Brinch Hansen 1970). I then used this paper as an outline of the 160-page system manual (Brinch Hansen 1969) by expanding each section of the paper.[7] The third article in this book is a reprint of the most important part of the original manual, which has been out of print for decades.[8]

As usual, Niklaus Wirth (1969) immediately recognized the advantages and limitations of the system:

> I am much impressed by the clarity of the multiple process concept, and even more so by the fact that a computer manufacturer adopts it as the basis of one of his products. I have come to the same conclusion with regard to semaphores, namely that they are not suitable for higher level languages. Instead, the natural synchronization events are exchanges of messages.
>
> What does not satisfy me completely at your scheme is that a specific mechanism of dynamic buffer space allocation is inextricably connected with the problem of process synchronization, I would prefer a scheme where the programmer himself declares such buffers in his programs (which of course requires an appropriate language).

4 Hierarchical Ordering of Sequential Processes

E. W. Dijkstra, Hierarchical Ordering of Sequential Processes (1971)

[7]In May 1968 I outlined these ideas in a panel discussion on Operating Systems at the Tenth Anniversary Algol Colloquium in Zurich, Switzerland. The panelists included Edsger Dijkstra and Niklaus Wirth, both of whom received copies of the RC 4000 system manual in July 1969.

[8]My operating system book (Brinch Hansen 1973b) includes a slightly different version of the original manual supplemented with abstract (untested) Pascal algorithms.

With deep insight, Dijkstra explains his *layered approach* to operating system design in greater detail. This time he proves the correctness of critical sections and the bounded buffer implemented with *semaphores*. He also introduces and solves the scheduling problem of the *dining philosophers*, which poses subtle dangers of *deadlock* and *unfairness* (described in flamboyant terminology as "deadly embrace" and "starvation").

The THE multiprogramming system was implemented in assembly language without memory protection. Every process could potentially access and change *any* variable in the system. However, using well-defined programming rules and systematic testing, Dijkstra and his students were able to verify that *all processes cooperated harmoniously.*

At the end of the paper, Dijkstra briefly sketches an alternative scenario of *secretaries* and *directors*:[9]

> Instead of N sequential processes cooperating in critical sections via common variables, we take out the critical sections and combine them into a N+1st process, called a "secretary"; the remaining N processes are called "directors". Instead of N equivalent processes, we now have N directors served by a common secretary.
>
> What used to be critical sections in the N processes are in the directors "calls upon the secretary".
>
> 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".
>
> In general, a director will like to send a message to his secretary when calling her ... and will require an answer back from his secretary when she has released his call.

On the basis of this proposal, Greg Andrews (1991) credits Dijkstra with being "the first to advocate using data encapsulation to control access to shared variables in a concurrent program." Twenty-five years ago, I repeated the prevailing opinion that "Dijkstra (1971) suggested the idea of monitors" (Brinch Hansen 1975a). Today, after reading the classic papers again, I find this claim (which Dijkstra never made) debatable.

[9]The gender bias in the terminology was not considered unusual thirty years ago.

Dijkstra had implemented his multiprogramming system as cooperating processes communicating through *common variables* in *unprotected memory.* From his point of view, the idea of combining critical regions and common variables into server processes ("secretaries") was a new approach to resource scheduling.

However, this idea was obvious to the designers of the RC 4000 multiprogramming system, based, as it was, on a paradigm of processes with *disjoint memories* communicating through *messages* only. There was simply no other way of using the RC 4000 system!

The "secretaries," which Dijkstra described informally, had already been implemented as "conversational processes" in the RC 4000 system. Mike McKeag (1972) demonstrated the similarity of these ideas by using the RC 4000 message primitives to outline simple secretaries for well-known synchronization problems, such as the *bounded buffer*, the *dining philosophers*, and a *readers and writers* problem.

I am not suggesting that the RC 4000 primitives would have been a good choice for a programming language. They would not. They lacked a crucial element of language design: notational elegance. And I certainly did not view conversational processes (or "secretaries") as the inspiration for the future monitor concept.

I am simply pointing out that the *idea* of a *resource manager* was already known by 1969, in the form of a *basic monitor*, invoked by supervisor calls, or a *conversational process* (a "secretary"), invoked by message passing.

What was new, was the goal of extending programming languages with this paradigm (Discussions 1971). And that had not been done yet.

PART II PROGRAMMING LANGUAGE CONCEPTS

The invention of *precise terminology* and *notation* plays a major role not only in the sciences but in all creative endeavors.

When a *programming concept* is understood informally it would seem to be a trivial matter to invent a *programming notation* for it. But in practice this is hard to do. The main problem is to replace an intuitive, vague idea with a precise, unambiguous definition of its meaning and restrictions. The mathematician George Pólya (1957) was well aware of this difficulty:

> An important step in solving a problem is to choose the notation. It should be done carefully. The time we spend now on choosing the notation may well be repaid by the time we save later by avoiding

> hesitation and confusion. Moreover, choosing the notation carefully, we have to think sharply of the elements of the problem which must be denoted. Thus, choosing a suitable notation may contribute essentially to understanding the problem.

A *programming language concept* must represent a *general idea* that is used often. Otherwise, it will just increase the complexity of the language at no apparent gain. The meaning and rules of a programming concept must be *precisely defined*. Otherwise, the concept is meaningless to a programmer. The concept must be represented by a *concise notation* that makes it easy to recognize the elements of the concept and their relationships. Finally, it should be possible by simple techniques to obtain a *secure, efficient implementation* of the concept. A compiler should be able to check that the rules governing the use of the concept are satisfied, and the programmer should be able to predict the speed and size of any program that uses the concept by means of performance measurements of its implementation.

As long as nobody studies your programs, their *readability* may not seem to be much of a problem. But as soon as you write a description for a wider audience, the usefulness of an *abstract notation* that suppresses irrelevant detail becomes obvious. So, although Dijkstra's THE system was implemented in assembly language, he found it helpful to introduce a programming notation for parallel statements in his description (Dijkstra 1965).

5 Conditional Critical Regions

In the fall of 1971, Tony Hoare enters the arena at a Symposium on Operating Systems Techniques at Queen's University of Belfast:

C. A. R. Hoare, Towards a Theory of Parallel Programming (1971)

This is *the first notable attempt to extend programming languages with abstract features for parallel programming*. Hoare points out that the search for parallel language features is "one of the major challenges to the invention, imagination and intellect of computer scientists of the present day."

Hoare boldly formulates *design principles* for *parallel programming languages*:

1. *Interference control*. The idea of preventing time-dependent errors by compile-time checking was novel at a time when multiprogramming systems relied exclusively on run-time checking of variable access:

> Parallel programs are particularly prone to time-dependent errors, which either cannot be detected by program testing nor by run-time checks. It is therefore very important that a high-level language designed for this purpose should provide complete security against time-dependent errors by means of a *compile-time check*.

2. *Disjoint processes.* Dijkstra's parallel statement $\{Q_1//Q_2//\ldots//Q_n\}$ is used to indicate that the program statements Q_1, Q_2, …, Q_n define disjoint processes to be executed in parallel. According to Hoare:

> It is expected that the compiler will check the disjointness of the processes by ensuring that no variable subject to change in any of the Q_j is referred to at all in any Q_i for $i \neq j$. Thus it can be guaranteed by a compile-time check that no time-dependent errors could ever occur at run time.

3. *Resources.* The programming language *Pascal* is extended with a notation indicating that a variable *r* of some type *T* is a resource shared by parallel processes:

$$\texttt{r: T; \ldots \{resource r; } Q_1//Q_2//\ldots//Q_n\}$$

4. *Critical regions.* Inside the process statements Q_1, Q_2, …, Q_n a critical region `C` on the resource `r` is expressed by the structured notation

```
with r do C
```

A compiler is expected to check that the resource is neither used nor referred to outside its critical regions.

5. *Conditional critical regions.* Sometimes the execution of a critical region `C` must be delayed until a resource `r` satisfies a condition, defined by a Boolean expression `B`:

```
with r when B do C
```

The conditional form of a critical region is the most original language feature proposed in Hoare's paper.[10]

[10] *Simula I* and *SOL* also included statements for waiting on Boolean conditions (later removed from Simula). However, these were simulation languages without any concept (or need) of critical regions (Dahl 1963, Knuth 1964).

```
B: record inpointer, outpointer, count: Integer;
   buffer: array 0...N-1 of T end;
 with B do
      begin inpointer:= 0; outpointer:= 0;
            count:= 0;
      end;
{resource B;
 ...
 with B when count < N do
      begin buffer[inpointer]:= next value;
            inpointer:= (inpointer + 1) mod N;
            count:= count + 1
      end
 //
 ...
 with B when count > 0 do
      begin this value:= buffer[outpointer];
            outpointer:= (outpointer + 1) mod N;
            count:=count - 1
      end
}
```

The Bounded Buffer with Conditional Critical Regions

Hoare emphasized that "The solutions proposed in this paper cannot claim to be final, but it is believed that they form a sound basis for further advance."

At the Belfast symposium (Brinch Hansen 1971a), I expressed some reservations from a software designer's point of view:

> 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.

Hoare's response (Discussions 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.
>
> A year ago I would have said that this was a very serious criticism indeed of a language proposal that it encouraged the programmer to ignore certain essential problems. I now believe that a language should be usable at a high level of abstraction, and at high levels of abstraction it is an excellent thing to encourage the programmer to ignore certain types of problems, in particular scheduling problems.

Hoare's paper was as an eye-opener for me: It was my introduction to *the difficult art of language design.* The idea of checking interference during scope analysis struck me as magical!

Years later, I included variants of conditional critical regions in two programming languages, *Distributed Processes* (Brinch Hansen 1978) and *Edison* (Brinch Hansen 1981).

6 Operating System Principles

Abstract concurrent programming had an immediate and dramatic impact on our fundamental understanding of computer operating systems.

The implementation techniques of operating systems were reasonably well understood in the late 1960s. But most systems were too large and poorly described to be studied in detail. All of them were written either in assembly language or in sequential programming languages extended with assembly language features. Most of the literature on operating systems emphasized low-level implementation details of particular systems rather than general concepts. The terminology was unsystematic and incomplete (Brinch Hansen 2000).

Before the invention of abstract concurrent programming, it was impractical to include algorithms in operating system descriptions. Technical writers mixed informal prose with unstructured flowcharts and complicated pictures of linked lists and state transitions.[11]

In its Cosine Report (1971), the National Academy of Engineering summarized the state of affairs at the time [with emphasis added]:

[11]See, for example, IBM (1965), Elliott Organick (1972), and Stuart Madnick (1974).

> The subject of computer operating systems, if taught at all, is typically a descriptive study of some specific operating system, with little attention being given to emphasizing the relevant basic concepts and principles. To worsen matters, *it has been difficult for most university departments to develop a new course stressing operating systems principles . . . There are essentially no suitable textbooks on the subject.*

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 at Carnegie-Mellon University, where I wrote *the first comprehensive textbook on operating system principles*:

P. Brinch Hansen, An Outline of a Course on Operating System Principles (1971)

While writing the book I reached the conclusion that *operating systems are not radically different from other programs. They are just large programs based on the principles of a more fundamental subject: parallel programming.*

Starting from a concise definition of the purpose of an operating system, I divided the subject into five major areas. First, I presented the principles of parallel programming as the essence of operating systems. Then I described processor management, memory management, scheduling algorithms and resource protection as techniques for implementing parallel processes.

I defined operating system concepts by abstract algorithms written in Pascal extended with a notation for structured multiprogramming. My (unimplemented) programming notation included *concurrent statements*, *semaphores*, *conditional critical regions*, *message buffers*, and *monitors*. These programming concepts are now discussed in all operating system texts.

The book includes a concise vocabulary of *operating system terminology*, which is used consistently throughout the text. The vocabulary includes the following terms:

> **concurrent processes**, *processes* that overlap in time; concurrent processes are called **disjoint** if each of them only refers to **private data**; they are called **interacting** if they refer to **common data**.
>
> **synchronization**, a general term for any constraint on the order in which *operations* are carried out; a synchronization rule can, for example, specify the precedence, priority, or mutual exclusion in time of operations.

> **monitor**, a *common data* structure and a set of meaningful *operations* on it that exclude one another in time and control the *synchronization* of *concurrent processes.*

My book *Operating System Principles* was published in July 1973. Peter Naur (1975) reviewed it:

> The presentation is generally at a very high level of clarity, and gives evidence of deep insight. In pursuing his general aim, the establishment of a coherent set of basic principles for the field, the author is highly successful. The principles are supported by algorithms written in Pascal, extended where necessary with carefully described primitives. Close attention is paid to the thorny question of terminology.

In my outline of the book I made a prediction that would guide my future research:

> 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.*

7 Structured Multiprogramming

P. Brinch Hansen, Structured Multiprogramming (1972)

The conditional critical region, proposed by Hoare (1971), had minor notational limitations and a potentially serious implementation problem:

1. A shared variable is declared as both a *variable* and a *resource.* The textual separation of these declarations can be misused to treat the same variable as a scheduled resource in some contexts and as an ordinary variable in other contexts. This would enable a process to refer directly to a variable while another process is within a "critical" region on the same variable.

I closed this loophole by using a single declaration to introduce a *shared variable* (of some type T):

var v: **shared** T

2. 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.*

It was an obvious remedy to permit a conditional *await statement* to appear anywhere within a critical region:

```
region v do
begin ... await B; ... end
```

3. The major concern was that *it did not seem possible to implement conditional critical regions efficiently.* The root of the problem is the unbounded reevaluation of Boolean expressions until they are true.

Many years later, Charles Reynolds (1993) asked:

> 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).

I suggested that programmers should be able to *associate secure queueing variables with shared data structures and control the transfers of processes to and from them.*

In my proposal, the declaration

```
var e: event v;
```

associates a queuing variable *e* of type *event* with a shared variable *v*.

A process can leave a critical region associated with *v* and join the queue *e* by executing the standard procedure

```
await(e)
```

```
var B: shared record
                buffer: array 0..max−1 of T;
                p, c: 0..max−1;
                full: 0..max;
                nonempty, nonfull: event B;
              end;
procedure send(m: T);
region B do
begin
  while full = max do await(nonfull);
  buffer[p] := m;
  p := (p + 1) mod max;
  full := full + 1;
  cause(nonempty);
end
procedure receive(var m: T);
region B do
begin
  while full = 0 do await(nonempty);
  m := buffer[c];
  c := (c + 1) mod max;
  full := full − 1;
  cause(nonfull);
end
```

The Bounded Buffer with Secure Events

Another process can enable all processes in the queue e to reenter their critical regions by executing the standard procedure

cause(e)

If several processes are waiting in the same queue, a *cause* operation on the queue will (eventually) enable *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 its scheduling condition B false again. Consequently, processes must use *waiting loops* of the form[12]

[12] *Mesa* (Lampson 1980) and *Java* (Lea 1997) would also require waiting loops on Boolean conditions.

while not B **do** await(e)

My proposal was completely unrelated to the unpredictable 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 secure scheduling queues with a shared data structure to enable processes to delay and resume critical regions has been used in all monitor proposals. In an unpublished draft, Hoare (1973a) proposed *wait* and *signal* operations on *condition variables*, which, he says, "are very similar to Brinch Hansen's await and cause operations." In the following I will call all these kinds of queues *secure queueing variables*.

Secure queueing variables were an efficient solution to the problem of process scheduling within critical regions. However, like semaphores, queueing variables always struck me (and others) as *somewhat too primitive* for abstract concurrent programming. To this day nobody has found a better compromise between notational elegance and efficient implementation. Still, I cannot help feeling that we somehow looked at the scheduling problem from the wrong point of view.

We now had all the pieces of the monitor puzzle, and I had adopted a programming style that combined shared variables, critical regions, secure queueing variables, and procedures in a manner that closely resembled monitors. But we still did *not* have an abstract monitor notation.

8 Shared Classes

The missing element in conditional critical regions was a concise representation of *data abstraction*. The declaration of a resource and the operations associated with it were not combined into a single syntactical form, but were distributed throughout the program text.

In the spring of 1972 I read two papers by Dahl (1972) and Hoare (1972) 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 system book:

P. Brinch Hansen, Shared Classes (1973)

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 a single monitor for a *bounded buffer*. The shared class defines a data structure of type B, two procedures that 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
```

The Bounded Buffer as a Shared Class

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

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 a buffer *b* are denoted

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

A shared class is a notation that explicitly restricts the operations on an *abstract 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 short, *a shared class is a monitor type*.

My decision to use *await statements* in the first monitor proposal was a matter of taste. I might just as well have used *secure queueing variables*.

You might well ask why after inventing shared classes with secure queueing variables I published my original ideas in a textbook, instead of a professional journal. Well, I was young and idealistic. I felt that my first book should include at least one original idea. It did not occur to me that researchers rarely look for original ideas in undergraduate textbooks.[13]

Why didn't I publish a tutorial on the monitor concept? My professional standards were deeply influenced by the *Gier Algol compiler* (Naur 1963), the *THE multiprogramming system* (Dijkstra 1968), the *RC 4000 multiprogramming system* (Brinch Hansen 1969), and the *Pascal compiler* (Wirth 1971). Every one of these systems had been implemented *before* it was described in a professional journal.

Since this was my standard of software research, I decided to implement monitors in a programming language before writing more about it.

9 Monitor Papers

In his first paper on monitors, Hoare (1973b) used my shared classes and secure queueing variables (with minor changes) to outline an unimplemented paging system. A year later, he published a second paper on monitors (Hoare 1974b). He acknowledged that "This paper develops Brinch Hansen's concept of a monitor."

Avi Silberschatz (1992) concluded that "The monitor concept was developed by Brinch Hansen (1973b). A complete description of the monitor was given by Hoare (1974b)."

[13]I did, however, send the complete manuscript of *Operating System Principles*, which included my monitor concept, to Edsger Dijkstra and Tony Hoare in May 1972 (Horning 1972).

C. A. R. Hoare, Monitors: An Operating System Structuring Concept (1974)

Hoare's contribution to the monitor concept was to *refine* the rules of *process resumption*:

1. He replaced the "resume-all, one-at-a-time" policy of secure event variables with the more convenient "first-come, first-served" policy of condition variables.

2. He decreed "that a signal operation be followed immediately by resumption of a waiting program, without possibility of an intervening procedure call from yet a third program." This eliminated the need for waiting loops.

3. He advocated Ole-Johan Dahl's simplifying suggestion that a signal operation should terminate a monitor call (Hoare 1973c).

```
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 ≤ count < N;
            buffer[lastpointer] := x;
            lastpointer := lastpointer ⊕ 1;
            count := count + 1;
            nonempty.signal
      end append;
    procedure remove(result x: portion);
      begin if count = 0 then nonempty.wait;
            note 0 < count ≤ N;
            x := buffer[lastpoint ⊖ count];
            count := count − 1;
            nonfull.signal
      end remove;
    count := 0; lastpointer := 0
  end bounded buffer;
```

The Bounded Buffer as a Monitor

This influential paper deserves a place in the history of concurrent programming as *the first monitor tutorial*:

1. The monitor concept is illustrated by solutions to familiar *programming exercises*: a single resource scheduler, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a readers and writers problem.

2. As an academic exercise he presents a *semaphore implementation* of monitors. (In practice, monitors would, of course, be implemented by uninterruptible operations in assembly language.)

3. Finally, he defines simple *proof rules* for condition variables.

PART III CONCURRENT PROGRAMMING LANGUAGES

Hoare (1974a) introduced the essential requirement that *a programming language must be secure* in the following sense: A language should enable its compiler and run-time system to detect as many cases as possible in which the language concepts break down and produce meaningless results.[14]

For a parallel programming language the most important security measure is to check that processes access disjoint sets of variables only and do not interfere with each other in time-dependent ways.

Unless the parallel features of a programming language are secure in this sense, the effects of parallel programs are generally both unpredictable and time-dependent and may therefore be meaningless. This does not necessarily prevent you from writing correct parallel programs. It does, however, force you to use a low-level, error-prone notation that precludes effective error checking during compilation and execution.

The only secret about secure concurrent languages was that they could be designed at all. Once you have seen that this is possible, it is not so difficult to invent other concurrent languages. That is why I have included only *the first secure concurrent language, Concurrent Pascal.*

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 1973e). The development of monitors and Concurrent Pascal started a wave of research in concurrent programming languages. A more recent survey of the field includes over 200 references to nearly 100 languages (Bal 1989).

Concurrent Pascal had obvious limitations by today's standards. But in 1975 it laid the foundation for the development of *secure programming languages with abstract concepts for parallelism.*

[14]This definition of *security* differs somewhat from its usual meaning of "the ability of a system to withstand attacks from adversaries" (Naur 1974).

10 Concurrent Pascal

On July 1, 1972, I became associate professor of computer science at 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. Since I had used the language in my operating system book, Pascal was an obvious choice for me.

In September 1973, I sent Mike McKeag "a copy of a preliminary working document that describes my suggestion for an extension of Pascal with concurrent processes and monitors" (Brinch Hansen 1973d). This is the earliest evidence of *Concurrent Pascal*.

By January 1975, the Concurrent Pascal compiler and its run-time support were running on a PDP 11/45 minicomputer at Caltech (Hartmann 1975, Brinch Hansen 1975f).

In May 1975, I published a paper on the new language:

P. Brinch Hansen, The Programming Language Concurrent Pascal (1975)

Concurrent Pascal extends Pascal with abstract data types known as processes, monitors, and classes. Each type module defines the representation and possible transformations of a single data structure. The syntax clearly shows that each module consists of a set of variable declarations, a set of procedures, and an initial statement.

A module cannot access the variables of another module. *The compiler uses this scope rule to detect synchronization errors before a program is executed.* The run-time synchronization of monitor calls prevents other race conditions.

A process can *delay* itself in a monitor variable of type *queue*. When another process performs a *continue* operation on the same queue, the delayed process (if any) immediately resumes execution of its monitor procedure. In any case, the process performing the continue operation immediately returns from its monitor procedure.

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

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*

```
type buffer =
monitor
var contents: array [1..max] of T;
      head, tail, length: integer;
      sender, receiver: queue;
procedure entry send(x: T);
begin
   if length = max then delay(sender);
   contents[tail] := x;
   tail := tail mod max + 1;
   length := length + 1;
   continue(receiver)
end;
procedure entry receive(var x: T);
begin
   if length = 0 then delay(receiver);
   x := contents[head];
   head := head mod max + 1;
   length := length − 1;
   continue(sender)
end;

begin head := 1; tail := 1; full := 0 end
```

The Bounded Buffer in Concurrent Pascal

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 somewhat more convenient to use.

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 slightly complicated. Fortunately, *monitors have the marvelous property of hiding the details of scheduling from concurrent processes.*

The programming tricks of assembly language were impossible in Concurrent Pascal: there were no typeless memory words, registers, and addresses in the language. The programmer was not even aware of the existence of physical processors and interrupts. *The language was so secure that concur-*

rent processes ran without any form of memory protection.

The portable compiler (written in Sequential Pascal) *generated platform-independent code, which was executed by a small kernel written in assembly language* (Hartmann 1975, Brinch Hansen 1975e). The language was moved from one computer to another by rewriting the kernel of 4K words in the assembly language of the target computer (Brinch Hansen 1975f).[15]

Greg Andrews (1993) felt that:

> 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.

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

> 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 (Brinch Hansen 1975d) suffered from all the problems I mentioned in the essay. I added, "I am particularly uncomfortable with the many *ad hoc* restrictions in the language."

Ole-Johan Dahl (1993) disagreed:

> 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.

[15]Twenty years later, the designers of the *Java* language resurrected the idea of platform-independent parallel programming (Gosling 1996). Unfortunately, they replaced the secure monitor concept of Concurrent Pascal with *insecure* shortcuts (Brinch Hansen 1999).

Concurrent Pascal was followed by more than a dozen *monitor languages*, listed in Brinch Hansen (1993a), among them *Modula* (Wirth 1977a), *Pascal Plus* (Welsh 1979), and *Mesa* (Lampson 1980).

PART IV MODEL OPERATING SYSTEMS

By the end of 1975, I had used Concurrent Pascal to implement three small operating systems of 600–1400 lines each:

- *The single-user operating system Solo*
- *A job stream system*
- *A real-time scheduler*

The development and documentation effort of each system took a few months (or weeks) only.

11 Solo Operating System

As a realistic test of the new programming language, I used Concurrent Pascal to program a small operating system:

P. Brinch Hansen, The Solo Operating System: A Concurrent Pascal Program (1976)

Solo was a portable single-user operating system for the development of Sequential and Concurrent Pascal programs. It was implemented on a PDP 11/45 minicomputer with removable disk packs. Every user disk was organized as a single-level file system. The heart of Solo was a job process that compiled and ran programs stored on the disk. Two additional processes performed input/output spooling simultaneously.

Al Hartmann (1975) had already written the Concurrent Pascal compiler. I wrote the operating system and its utility programs in three months. Wolfgang Franzen measured and improved the performance of the disk allocation algorithm.

The Solo system demonstrated that it is possible to write small operating systems in a secure programming language without machine-dependent features. The discovery that this was indeed possible for small operating systems was more important (I think) than the invention of monitors.

12 Solo Program Text

Solo was *the first modular operating system implemented by means of abstract data types (classes, monitors and processes) with compile-time checking of access rights.* The most significant contribution of Solo was undoubtedly that the program text was short enough to be *published in its entirety* in a computer journal:

P. Brinch Hansen, The Solo Operating System: Processes, Monitors and Classes (1976)

The new programming language had a dramatic (and unexpected) impact on my programming style. 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.* Since each module defined all the meaningful operations on a single data type (private or shared), the modules could be studied and tested one at a time. As a result these *concurrent programs became more reliable than the hardware they ran on.*

In July 1975, when the Solo operating system had been working for three months, I described it at the International Summer School in Marktoberdorf, Germany. Hoare presented an outline of an unimplemented operating system (Hoare 1976a).

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.

In a guest editorial on the Solo papers (Brinch Hansen 1976a), I wrote:

> It is not uncommon for a computer scientist to make a proposal without testing whether it is any good in practice. After spending 3 days writing up the monitor proposal and 3 years implementing it, I can very well understand this temptation. It is perhaps also sometimes a human response to the tremendous pressure on university professors to get funding and recognition fast.
>
> Nevertheless, we must remember that only one thing counts in engineering: Does it work (not "might it work" or "wouldn't it be nice

> if it did")? What would we think of mathematicians if most of their papers contained conjectures only? Sometimes an educated guess can be a great source of inspiration. But we must surely hope that the editors of computer journals will reject most proposals until they have been tried at least experimentally.
>
> All reviewers of my [operating system] book correctly pointed out that the chapter on resource protection [introducing shared classes] was highly speculative. The Solo operating system described here is an attempt to set the record straight by putting monitors to a realistic test.

13 The Architecture of Concurrent Programs

In July 1976 I joined University of Southern California as professor and chair of computer science. Now that Concurrent Pascal was running I knew that the time was ripe for a book on the principles of abstract parallel programming.

My second book, *The Architecture of Concurrent Programs*, includes the complete text of the model operating systems written in Concurrent Pascal (Brinch Hansen 1977b).

In a book review, Roy Maddux and Harlan 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.

Twenty years later, two of my former Ph.D. students recalled their experience of working with Concurrent Pascal:

> *Jon Fellows* (1993): "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."

> *Charles Hayden* (1993): "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 ... 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."

In designing Concurrent Pascal and the model operating systems written in the language I followed a consistent set of programming principles. These principles carried *structured programming* (Dijkstra 1972a) into the realm of *modular, concurrent programming*:

P. Brinch Hansen, Design Principles (1977)

Roy Maddux and Harlan Mills (1979) agreed that:

> An author does well to start by stating those beliefs and biases he holds that are relevant to his work so that the reader is forewarned about what will follow and can understand the motivation behind subsequent decisions and choices. Brinch Hansen's opening chapter—a reasoned essay on the fundamental principles of programming today—does this remarkably well. The quotations at the end of the chapter are particularly well-chosen and make delightful reading.

PART V DISTRIBUTED COMPUTING

In the late 1970s, parallel computing was moving from multiprocessors with shared memory towards *multicomputers* with *distributed memory.* For microcomputer networks, Dijkstra, Hoare and I suggested different programming models. Although our ideas opened the way for *abstract distributed computing*, they clearly needed further refinement before they could be incorporated into programming languages.

14 A Synthesis Emerging?

Edsger Dijkstra led the way. In a brief note he gave a personal account of a discussion with Tony Hoare at the International Summer School in Marktoberdorf, Germany, in July 1975:

E. W. Dijkstra, A Synthesis Emerging? (1975)

Hoare was trying to explain the class concept of Simula 67, when Dijkstra began to:

> change terminology, notation and a way of looking at it, things I had to do in order to make it all fit within my frame of mind. To begin with, I shall record how our discussions struck root in my mind. I don't know whether a real Simula fan will still recognize the class-concept; he may get the impression that I am writing about something totally different.

Indeed! What emerges is the exciting possibility of *modular programs with nondeterministic process types* (called *generators*). In his usual colorful terminology, Dijkstra calls these programs "elephants built from mosquitoes."

His simplest example is a generator, named *nn*, for *natural numbers*:

```
nn gen begin privar x; x virint := 0;
             do ?inc → x := x + 1
              ▯ x > 0 cand ?dec → x := x − 1
             od
       end
```

(The notational details are not important here.)

The main program can declare a variable y as a natural number:

```
privar y; y vir nn;
```

The *generator instance* y keeps a natural number in a *private variable* x. After initializing its value to zero, the generator is ready to perform an endless series of *increase* and *decrease* operations on x in response to *commands* from the main program:

```
y.inc        y.dec
```

The generator defines the increment operation as a *guarded command*

$$?\mathrm{inc} \rightarrow \mathrm{x} := \mathrm{x} + 1$$

When the main program issues an increment command, the *guard*

$$?\mathrm{inc}$$

is regarded as being true (once), enabling the generator to execute the *guarded statement*

$$\mathrm{x} := \mathrm{x} + 1$$

However, if the main program issues a decrement command, the guard

$$\mathrm{x} > 0 \ \mathbf{cand}\ ?\mathrm{dec}$$

does not become true until $x > 0$.

So far, the generator looks very much like a *monitor* implementation of a *semaphore*, but there are subtle differences:

- *Dijkstra views the main program and its generators as processes that are synchronized during the execution of guarded commands.*
- *When the main program terminates, all guards within its local generators become false, and the generator loops terminate too.*

Dijkstra emphasizes that:

> [In the past] it was the purpose of our programs to instruct our machines: now it is the purpose of the machines to execute our programs. Whether the machine does so sequentially, one thing at a time, or with a considerable amount of concurrency, is a matter of implementation and should *not* be regarded as a property of the programming language.

This viewpoint naturally leads him to conclude that

- If the main program is *concurrent*, the generator does indeed implement a *semaphore* that *delays* a decrement operation until $x > 0$.
- However, if the main program is *sequential*, an attempt to decrement a natural number equal to zero will cause the main program to get *stuck*.

At this point Dijkstra introduces the powerful concept of *recursive nondeterministic processes.* He programs a generator that defines a *sequence of integers* recursively. A parallel execution of this program can be visualized as *a pipeline of processes.* Each process accepts commands from its predecessor (which is either another pipeline process or the main program).

An *insert command*, issued by the main program, *propagates* to the end of the chain, where the last process *extends the pipeline* with another process.

A *membership query* moves down the pipeline until it either reaches a process that holds the desired element or is absorbed at the end of the pipeline. In a parallel implementation, a *wave of queries* can move down the pipeline simultaneously.

Edsger Dijkstra called it "A surprising discovery, the depth of which is as far as I am concerned still unfathomed." In 1982 he added a final remark:

> In retrospect this text is not without historical interest: it records the highlights of a discussion mentioned [as "Verbal communication" (Dijkstra 1975)] in C. A. R. Hoare's "Communicating sequential processes", Comm. ACM 21, 8 (Aug. 1978), 666-677. The text was evidently written in a state of some excitement; in retrospect we may conclude that this excitement was not entirely unjustified. Seeing Hoare keenly interested in the topic, I left that arena.

15 Communicating Sequential Processes

Three years after his discussion with Edsger Dijkstra in Marktoberdorf, Tony Hoare publishes a paper on *communicating sequential processes* (also known as *CSP*):

C. A. R. Hoare, Communicating Sequential Processes (1978)

This classic paper develops Dijkstra's (1975a) vision of nondeterministic processes communicating by means of guarded commands (but *without recursion*).

The *bounded buffer*, shown here, is a CSP *process*, named `X`, that can hold up to ten buffer portions. After making the buffer empty to begin with, the process executes a *repetitive command* (prefixed by an asterisk `*`). In each cycle, the buffer process is delayed until one of two possible communications takes place:

1. A process named `producer` is ready to execute an *output command* `X!e`. In that case, the buffer process inputs the value of the expression

```
X::
buffer:(0..9)portion;
in,out:integer; in:= 0; out:= 0;
comment 0 ≤ out ≤ in ≤ out + 10;
  ∗[in < out + 10; producer?buffer(in mod 10) → in:= in + 1
   []out < in; consumer?more() → consumer!buffer(out mod 10);
    out := out + 1
  ]
```

The Bounded Buffer in CSP

`e` in the last buffer element, provided that there is room for it in the buffer. This is the effect of the *guarded input command*:

```
in < out + 10; producer?buffer(in mod 10) → in:= in + 1
```

2. A process named `consumer` outputs a request for more input, `X!more()`, and inputs the next buffer portion in a local variable `v` by executing the command `X?v`. When the buffer is nonempty, it accepts the request before outputting the first portion:

```
out < in; consumer?more() →
  consumer!buffer(out mod 10); out:= out + 1
```

This paper describes *highly original ideas*:

1. *Synchronous communication.* Hoare introduces this idea, which was well-known in computer architectures but novel in programming languages:

> Communication occurs when one process names another as destination for output *and* the second process names the first as source for input. In this case, the value to be output is copied from the first process to the second. There is *no* automatic buffering: In general, an input or output command is delayed until the other process is ready with the corresponding output or input. Such delay is invisible to the delayed process.

2. *Input guards.* CSP incorporates Dijkstra's (1975a) concept of nondeterministic process interactions controlled by guarded commands:

> A guarded command with an input guard is selected for execution only if and when the source named in the input command is ready to execute the corresponding output command. If several input guards of a set of alternatives have ready destinations, only one is selected and the others have *no* effect; but the choice between them is arbitrary.

3. *Coincidence of events.* In 1965, Dijkstra demonstrated that *mutual exclusion* of events is a fundamental programming concept. In 1975, he showed that the opposite idea, the *coincidence* of events, is just as important! This strikes me as the most profound idea incorporated in CSP.

4. *Programming examples.* The CSP paper includes solutions to a wide variety of interesting problems.

However, the CSP proposal also has some *awkward details*:

1. *Direct process naming.* One of the major advantages of monitors is their ability to communicate with processes and schedule them without being aware of process identities. In CSP, an input/output command must name the source or destination process directly. The text of a process must therefore be modified when it is used in different contexts. This complicates the examples in Hoare's paper: the user of a process array `S(1..n)` is itself named `S(0)`. And the prime sieve is composed of three different kinds of processes to satisfy the naming rules.

2. *Pattern matching.* The CSP notation does not include type declarations of communication channels, but depends (conceptually) on *dynamic checking* to recognize matching input and output commands in parallel processes.

3. *Conditional input.* Hoare mentions that:

> conditions can be used to delay acceptance of inputs which would violate scheduling constraints—postponing them until some later occasion when some other process has brought the monitor into a state in which the input can validly be accepted. This technique is similar to a conditional critical region (Hoare 1971) and it obviates the need for special synchronizing variables such as events, queues, or conditions. However, the absence of these special facilities certainly makes it more difficult or less efficient to solve problems involving priorities.[16]

[16]Notice, however, that a *monitor* with *await statements* on Boolean conditions does not require queueing variables either (Brinch Hansen 1973c).

4. *No output guards.* This restriction forces Hoare to publish a CSP version of the bounded buffer with *asymmetric input/output operations.* For aesthetic reasons, I find this lack of elegance regrettable.

5. *Process termination.* CSP uses Dijkstra's (1975a) termination rule:

> A repetitive command may have input guards. If all the sources named by them have terminated, then the repetitive command also terminates.

Hoare maintains that:

> The automatic termination of a repetitive command on termination of the sources of all its input guards is an extremely powerful and convenient feature but it also involves some subtlety of specification to ensure that it is implementable; and it is certainly not primitive, since the required effect can be achieved (with considerable inconvenience) by explicit exchange of "end()" signals.

Seven years later, Hoare (1985) realizes that:

> The trouble with this convention is that it is complicated to define and implement; and methods of proving program correctness seem no simpler with it than without.

6. *No recursion.* The most obvious weakness of CSP is the *omission of* Dijkstra's beautiful concept of *recursive nondeterministic processes.* A CSP process cannot activate itself recursively. It is, however, possible to activate *fixed-length process arrays*, which can imitate the behavior (but not the elegance) of recursive processes.[17]

CSP was a major achievement and the inspiration for a new generation of concurrent programming languages, including the nonrecursive language *occam* for the *transputer* (Inmos 1989a, 1989b) and the recursive language *Joyce* (Brinch Hansen 1987a).

Seven years later Hoare (1985) published a *mathematical theory of communicating sequential processes* using a recursive variant of CSP. This notation has played a significant role in research on the mathematical foundations of concurrency. Hoare (1981) is an early example of this theoretical work (which is beyond the scope of this essay).

[17]My alternative programming model, *Distributed Processes*, is also nonrecursive (Brinch Hansen 1978c).

16 Distributed Processes

For microcomputer networks with distributed memory I introduced the idea of *a synchronized procedure that can be called by one process and executed by another process.* This proposal *combines processes and monitors into a single concept*, called *distributed processes.* In distributed operating systems, this communication paradigm is known as *remote procedure calls.*

P. Brinch Hansen, Distributed Processes: A Concurrent Programming Concept (1978)

Distributed Processes have the following properties:

- A real-time program consists of a fixed number of concurrent processes that are started simultaneously and exist forever. Each process can access its *own variables* only. There are no common variables.
- A process can call *common procedures* defined within other processes. These procedures are executed when the other processes are waiting for some conditions to become true. A procedure call from one process to another is called an *external request.* This is the only form of process communication.
- Processes are synchronized by means of nondeterministic *guarded regions* (Hoare 1971, Dijkstra 1975b, Brinch Hansen 1978c).

The *bounded buffer*, shown here, is a process that stores a sequence of characters transmitted between processes by means of *send* and *receive* procedures.

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

The Bounded Buffer with Distributed Processes

The initial statement makes the buffer empty and terminates. The buffer process, however, continues to exist and can now be called by other processes:

call buffer.send(e) **call** buffer.rec(v)

After initialization, the buffer process is idle between *external calls.* This process is similar to a *monitor* (Brinch Hansen 1973c) with *conditional critical regions* (Hoare 1971).

In general, an external call of a procedure R, declared in a process Q, may include both *value* and *result parameters*:

call $Q.R$(expressions, variables)

The *parameter passing* between two distributed processes requires a single *input* operation when an external procedure is activated, followed by a single *output* operation when it terminates.

The relationship between two communicating processes is *asymmetrical* and requires only that the caller of a procedure name the process that performs it. This asymmetry is useful in *hierarchical systems*, in which *server processes* should be unaware of the identities of *client processes.*

Every process is *quasiparallel* in the following sense:

- A process begins by executing its initial statement. This continues until the statement either terminates or waits for a condition to become true. Then another operation is started (as the result of an external request). When this operation in turn terminates or waits the process will either begin yet another operation (requested by another process) or it will resume an earlier operation (as the result of a condition becoming true). This interleaving of the initial statement and the external requests continues forever. If the initial statement terminates, the process continues to exist and will still accept external statements.
- In a microprocessor network where each processor is dedicated to a single process it is an attractive possibility to let a process carry out computations *between* external calls of its procedures. The *shortest job next scheduler* (shown in the paper) takes advantage of this capability by selecting the next user while the resource is being used by the present user.

The major *weaknesses* of distributed processes are (1) the implicit *waiting loops* on Boolean conditions and (2) the *absence of parallel recursion.*

It was Jim White (1976) who first proposed remote procedure calls, as an *informal programming style.* However, White did not explain how to prevent *race conditions* between unsynchronized remote calls and local processes that are being executed by the same processor. This flaw potentially made remote procedure calls as unsafe as interrupts that cannot be disabled! Nevertheless, the original idea was his.

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

Greg Andrews (1991) acknowledged that:

> Per Brinch Hansen (1978) developed the first programming language based on [remote procedure calls] RPC. His language is called Distributed Processes (DP).[18]

According to Olivier Roubine (1980), my proposal was "a source of inspiration in the design of the Ada tasking facilities." The *rendezvous* concept in the language *Ada* combines the remote procedure calls of distributed processes with the selection of alternative interactions in CSP.

Since then, operating system designers have turned remote procedure calls into an *unreliable mechanism* of *surprising complexity.* In their present form, remote procedure calls are an attempt to use *unreliable message passing* to invoke procedures through local area networks.

Tay (1990) admits that "Currently, there are no agreed definition on the semantics of RPC." Leach (1983) goes one step further and advocates that "each remote operation implements a protocol tailored to its need." Since it can be both *system-dependent* and *application-dependent*, a remote procedure call is no longer an abstract concept.

After implementing a remote procedure call mechanism for the distributed operating system *Unix United*, Santosh Shrivastava and Fabio Panzieri (1982) concluded:

> At a superficial level it would seem that to design a program that provides a remote procedure call abstraction would be a straightforward exercise. Surprisingly, this is not so. We have found the problem of the design of the RPC to be rather intricate.

[18]Rarely does anyone replace single words, like "Pascal," "Monitor," "Solo" or "Joyce," by baffling acronyms—P, M, S or J. But carefully chosen longer names, like "Conditional Critical Region," "Concurrent Pascal," "Communicating Sequential Processes," "Distributed Processes" and "Remote Procedure Call," are doomed to be abbreviated as CCR, CP, CSP, DP and RPC. *If you believe that papers should be easy to read (but not necessarily easy to write), the lesson is clear: Always use single words to name your concepts!*

17 Joyce

The most surprising idea in Dijkstra's "Emerging Synthesis" (1975a) was his introduction of *recursive nondeterministic processes.* This idea was clearly ahead of its time. Some ten years would pass before Hoare (1985) published a theoretical recursive variant of CSP.

Two years later, I published the first *recursive CSP language* implemented on a computer:

P. Brinch Hansen, Joyce—A Programming Language for Distributed Systems (1987)

Joyce is a secure CSP language based on a minimal subset of Pascal. A Joyce program activates *recursive processes*, known as *agents.* These agents communicate through *synchronous channels.* A channel can transfer messages of different (but fixed) types between two or more agents. The compiler checks *message types* and ensures that agents use *disjoint variables* only.

```
type stream = [int(integer)];

agent buffer(inp, out: stream);
const n = 10;
type contents = array [1..n] of integer;
var head, tail, length: integer;
  ring: contents;
begin
  head := 1; tail := 1; length := 0;
  while true do
    poll
      inp?int(ring[tail]) & length < n ->
        tail := tail mod n + 1;
        length := length + 1|
      out!int(ring[head]) & length > 0 ->
        head := head mod n + 1;
        length := length − 1
    end
end;
```

The Bounded Buffer in Joyce

The *bounded buffer*, shown here, is defined by an *agent procedure.* A buffer agent uses two channels of type *stream.* Every communication through a stream channel transmits a single *symbol*, named *int*, from one agent to another. The symbol carries a *message* of type integer.

A buffer agent transmits an endless stream of symbols from one channel to another in response to *input/output commands* from other agents. In each cycle, a buffer agent executes a *polling statement* that delays it until a *conditional communication* takes place through one of its channels.

In general:

> A Joyce program consists of nested procedures which define communicating agents. Joyce permits unbounded (recursive) activation of agents. The execution of a program activates an initial agent. Agents may dynamically activate subagents which run concurrently with their creators. The variables of an agent are inaccessible to other agents.
>
> Agents communicate by means of symbols transmitted through channels. Every channel has an alphabet—a fixed set of symbols that can be transmitted through the channel. A symbol has a name and may carry a message of a fixed type.
>
> Two agents match when one of them is ready to output a symbol to a channel and the other is ready to input the same symbol from the same channel. When this happens, a communication takes place in which a message from the sending agent is assigned to a variable of the receiving agent.
>
> The communications on a channel take place one at a time. A channel can transfer symbols in both directions between two agents.
>
> A channel may be used by two or more agents. If more than two agents are ready to communicate on the same channel, it may be possible to match them in several different ways. The channel arbitrarily selects two matching agents at a time and lets them communicate.
>
> A polling statement enables an agent to examine one or more channels until it finds a matching agent. Both sending and receiving agents may be polled.
>
> Agents create channels dynamically and access them through local port variables. When an agent creates a channel, a channel pointer is assigned to a port variable. The agent may pass the pointer as a parameter to subagents.
>
> When an agent reaches the end of its defining procedure, it waits until all its subagents have terminated before terminating itself. At this point, the local variables and any channels created by the agent cease to exist.

Hoare (1978) emphasized that CSP should not be regarded as suitable for use as a programming language but only as a partial solution to the problems tackled.

Joyce removed unnecessary limitations of CSP by introducing:

- Recursive agent procedures.
- Simple agent termination.
- Typed channel alphabets.
- Typed port variables.
- Bidirectional synchronous channels.
- Nondeterministic shared channels.
- Symmetric input/output polling.

To be able to experiment with parallel recursion, I developed *portable implementations* of Joyce for a *personal computer* and a *multiprocessor* (Brinch Hansen 1987b, 1989b).

I still marvel at the beauty of recursive agents, such as the *bounded buffer*, the *sorting array*, the *prime sieve*, the *integer set*, and the *Fibonacci tree* (shown in the paper).

How can I explain the joy of being able, for the first time, to explore this new class of algorithms in a concise, executable language? The experience reminds me of the wise observation by the logician Susanne K. Langer (1967):

> There is something uncanny about the power of a happily chosen ideographic language; for it often allows one to express relations which have no names in natural language and therefore have never been noticed by anyone. Symbolism, then, becomes an organ of discovery rather than mere notation.

PART VI IMPLEMENTATION ISSUES

I promised to omit "Implementation details (except in outline)." Parallel programming languages do, however, pose special implementation problems that deserve your attention:

- *Interference control during compilation.*
- *Memory allocation of parallel recursion.*

18 SuperPascal

When Hoare (1971) published his paper on conditional critical regions, we did not fully appreciate the complexity of checking interference in a block-structured parallel language. You see, the subsequent invention of *modular parallelism* made interference checking so simple that we hardly noticed how hard it could have been!

Out of curiosity I asked myself twenty-three years later, *Is it feasible to detect process interference in a block-structured language with nonmodular parallelism*?

P. Brinch Hansen, SuperPascal—A Publication Language for Parallel Scientific Computing (1994)

The parallel features of *SuperPascal* are a subset of *occam 2* with the added generality of *dynamic process arrays* and *recursive parallel processes* (Inmos 1988b, Cok 1991). SuperPascal omits ambiguous and insecure features of Pascal. *Restrictions on the use of variables enable a single-pass compiler to check that parallel processes are disjoint, even if the processes use procedures with global variables.*[19]

When you have read this paper, you can judge for yourself how complicated concurrent programming would have been without some form of modularity, such as the process and monitor types of Concurrent Pascal.

After reading the paper, Dave Parnas (1993) felt that "Some might suggest that nobody would be able to build practical programs in a language with so many restrictions." I answered (Brinch Hansen 1993d):

> I too was surprised at the restrictions required to make parallelism secure in a block-structured language. However, I think that the exercise merely forced me explicitly to recognize the complexity of the procedure concept in our programming languages (such as Pascal). SuperPascal forced me to use a more restricted procedure concept. So far, I have found that the rules enforced by the compiler contribute to program clarity.

After developing a portable implementation of SuperPascal on a Sun workstation:

[19]Since the language does not support conditional communication, a bounded buffer cannot be programmed in SuperPascal.

> [I used] the SuperPascal notation to write portable programs for regular problems in computational science (Brinch Hansen 1995). I found it easy to express these programs in three different programming languages (SuperPascal, Joyce,[20] and occam 2) and run them on three different architectures (a Unix workstation, an Encore Multimax, and a Meiko Computing Surface).[21]

19 Efficient Parallel Recursion

In *CSP* and *Distributed Processes*, Hoare and I shied away from parallel recursion because of the difficulty of *implementing an unbounded tree-structured stack without using garbage collection.*

Dijkstra (1975a) was well aware of this stumbling block:

> the storage requirements for a sequence are very simple, viz. a stack. (In our rejected example of the binary tree, although lifetimes are, in a fashion, nested, life is not so simple.)

After using static memory allocation in Concurrent Pascal, it took me twenty years to discover a simple method for *efficient parallel recursion* (which I used to implement *SuperPascal*):

P. Brinch Hansen, Efficient Parallel Recursion (1995)

I now believe that *we should have used parallel recursion from the beginning, even though we didn't know how to implement it.*[22] This kind of intellectual courage paid off handsomely when Peter Naur (1960) included *sequential recursion* in his famous *Algol 60* report, *before* Dijkstra (1960) had shown how to implement it efficiently using a run-time stack.

THE END OF AN ERA

The development of *abstract language notation for concurrent programming* started in 1965. Twenty years later Judy Bishop (1986) concluded:

[20] Brinch Hansen (1988).

[21] The Encore Multimax was a *multiprocessor* with 18 processors sharing a memory of 128 MB (Trew 1991). The Computing Surface was a *multicomputer* with 48 transputers, each with 1 MB of local memory (Inmos 1988a, Trew 1991).

[22] As you can tell, I am now a middle-aged idealist.

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.

So passed an exciting era.

Acknowledgements

It is a pleasure to acknowledge the perceptive comments of Charles Hayden, Henk Kruijer, Peter O'Hearn and Charles Reynolds.

I thank the Association for Computing Machinery, the Institute of Electrical and Electronics Engineers, and Springer-Verlag for permission to include parts of my earlier essays:

- P. Brinch Hansen, A keynote address on concurrent programming. *Computer 12*, 5 (May 1979), 50–56. Copyright © 1979, Institute of Electrical and Electronics Engineers, Inc.
- 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.
- P. Brinch Hansen, The evolution of operating systems. In P. Brinch Hansen Ed., *Classic Operating Systems: From Batch Processing to Distributed Systems.* Springer-Verlag, New York, 2001, 1–34. Copyright © 2001, Springer-Verlag New York, Inc.

BIBLIOGRAPHY

⋆ *Classic* papers included in this book.
• *Recommended* for further reading.
○ *Cited* for historical reasons.

1. • G. R. Andrews and F. B. Schneider 1983. Concepts and notations for concurrent programming. *Computing Surveys 15*, 1 (March), 3–43.
2. • G. R. Andrews 1991. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, CA.
3. ○ G. R. Andrews 1993. Reviewers' comments. In P. Brinch Hansen, Monitors and Concurrent Pascal: a personal history. *SIGPLAN Notices 28*, 3 (March 1993), 1–35.
4. • H. E. Bal, J. G. Steiner and A. S. Tanenbaum 1989. Programming languages for distributed computing systems. *ACM Computing Surveys 21*, (September), 261–322.
5. • J. Bishop 1986. *Data Abstraction in Programming Languages.* Addison-Wesley, Reading, MA.

6. • J. Boyle, R. Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson and Rick Stevens 1987. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, New York.

7. ◦ P. Brinch Hansen 1967a. The logical structure of the RC 4000 computer. *BIT 7*, 3, 191–199.

8. ◦ P. Brinch Hansen 1967b. The RC 4000 real-time control system at Pulawy. *BIT 7*, 4. 279–288.

9. ◦ P. Brinch Hansen 1968. *The Structure of the RC 4000 Monitor.* Regnecentralen, Copenhagen, Denmark (February).

10. ⋆ P. Brinch Hansen 1969. *RC 4000 Software: Multiprogramming System.* Regnecentralen, Copenhagen, Denmark, (April). *Article 3.*

11. • P. Brinch Hansen 1970. The nucleus of a multiprogramming system. *Communications of the ACM 13*, 4 (April), 238–241, 250.

12. ⋆ P. Brinch Hansen 1971a. An outline of a course on operating system principles. In C. A. R. Hoare and R. H. Perrott Eds. 1972, *Operating Systems Techniques*, Proceedings of a Seminar at Queen's University, Belfast, Northern Ireland, August–September 1971, Academic Press, New York, 29–36. *Article 6.*

13. • P. Brinch Hansen 1972a. A comparison of two synchronizing concepts. *Acta Informatica* **1**, 3 (1972), 190–199.

14. ⋆ P. Brinch Hansen 1972b. Structured multiprogramming. *Communications of the ACM 15*, 7 (July), 574–578. *Article 7.*

15. • P. Brinch Hansen 1973a. Testing a multiprogramming system. *Software—Practice and Experience 3*, 2 (April–June), 145–150.

16. • P. Brinch Hansen 1973b. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, NJ (May)..

17. ⋆ P. Brinch Hansen 1973c. Class Concept. In *Operating System Principles*, P. Brinch Hansen, Prentice Hall, Englewood Cliffs, NJ, (July 1973), 226–232. *Article 8.*

18. ◦ P. Brinch Hansen 1973d. 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" (R. M. McKeag, letter to P. Brinch Hansen, July 3, 1991). No longer available.

19. ◦ P. Brinch Hansen 1973e. Concurrent programming concepts. *ACM Computing Surveys 5*, 4, (December), 223–245.

20. ◦ P. Brinch Hansen 1974a. Concurrent Pascal: a programming language for operating system design. Information Science, California Institute of Technology, Pasadena, CA, (April). (Referenced in A. Silberschatz, R. B. Kieburtz and A. J. Bernstein 1977. Extending Concurrent Pascal to allow dynamic resource management. *IEEE Transactions on Software Engineering 3*, (May), 210–217.)

21. ⋆ P. Brinch Hansen 1975a. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering 1*, 2 (June), 199–207. Original version: Information Science, California Institute of Technology, Pasadena, CA, (November 1974). *Article 10.*

22. ⋆ P. Brinch Hansen 1975b. The Solo operating system: a Concurrent Pascal program. Information Science, California Institute of Technology, Pasadena, CA, (June–July). Also in *Software—Practice and Experience 6*, 2 (April–June 1976), 141–149. *Article 11.*

23. ⋆ P. Brinch Hansen 1975c. The Solo operating system: processes, monitors and classes. Information Science, California Institute of Technology, Pasadena, CA, (June–July). Also in *Software—Practice and Experience 6*, 2 (April–June 1976), 165–200. *Article 12.*

24. ○ P. Brinch Hansen 1975d. Concurrent Pascal report. Information science, California Institute of Technology, Pasadena, CA, (June). Also in P. Brinch Hansen 1977b, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, NJ, (July), 231–270.

25. ○ P. Brinch Hansen and A. C. Hartmann 1975e. Sequential Pascal report. Information science, California Institute of Technology, Pasadena, CA, (July).

26. ○ P. Brinch Hansen 1975f. Concurrent Pascal machine. Information Science, California Institute of Technology, Pasadena, CA, (October). Also in P. Brinch Hansen 1977b, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, NJ, (July), 271–297.

27. ○ P. Brinch Hansen 1976a. Innovation and trivia in program engineering. Guest Editorial, *Software—Practice and Experience 6*, 2 (April–June), 139–140.

28. ○ P. Brinch Hansen 1976b. Concurrent Pascal implementation notes. Information Science, California Institute of Technology, Pasadena, CA. No longer available. (Referenced in M. S. Powell, Experience of transporting and using the Solo operating system. *Software—Practice and Experience 9*, 7 (July 1979), 561–570.)

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

30. • P. Brinch Hansen 1977b. *The Architecture of Concurrent Programs.* Prentice-Hall, Englewood Cliffs, NJ, (July).

31. ⋆ P. Brinch Hansen 1977c. Design Principles. In P. Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice Hall, Englewood Cliffs, NJ, (July), 3–14. *Article 13.*

32. • P. Brinch Hansen 1978a. Network: A multiprocessor program. *IEEE Transactions on Software Engineering 4*, 3 (May), 194–199.

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

34. ⋆ P. Brinch Hansen 1978c. Distributed Processes: a concurrent programming concept. *Communications of the ACM 21*, 11 (November), 934–941. *Article 16.*

35. • P. Brinch Hansen 1978d. A keynote address on concurrent programming. *IEEE Computer Software and Applications Conference*, Chicago, IL, (November), 1–6. Also in *Computer 12*, 5 (May 1979), 50–56.

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

37. • P. Brinch Hansen 1979. The end of a heroic era. In P. Wegner Ed. *Research Directions in Software Technology*, MIT Press, Cambridge, MA, 646–649.

38. • P. Brinch Hansen and J. A. Fellows 1980. The Trio operating system. *Software—Practice and Experience 10*, 11 (November), 943–948.

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

40. ⋆ P. Brinch Hansen 1987a. Joyce—A programming language for distributed systems. *Software—Practice and Experience 17* , 1 (January), 29–50. *Article 17.*

41. • P. Brinch Hansen 1987b. A Joyce implementation. *Software—Practice and Experience 17*, 4 (April 1987), 267–276.

42. ○ P. Brinch Hansen and A. Rangachari 1988. Joyce performance on a multiprocessor. School of Computer and Information Science. Syracuse University, NY, (September).

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

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

45. • P. Brinch Hansen 1990. The nature of parallel programming. In M. A. Arbib and J. A. Robinson Eds., *Natural and Artificial Parallel Computation*, MIT Press, Cambridge, MA, 31–46.

46. • P. Brinch Hansen 1993a. Monitors and Concurrent Pascal: a personal history. *SIGPLAN Notices 28*, 3 (March), 1–35. Also in T. J. Bergin and R. G. Gibson Eds. 1996, *History of Programming Languages II*, Addison-Wesley Publishing, Reading, MA, 121–172.

47. • P. Brinch Hansen 1993b. Model programs for computational science: a programming methodology for multicomputers. *Concurrency—Practice and Experience 5*, 5 (August), 407–423.

48. • P. Brinch Hansen 1993c. Parallel cellular automata: a model program for computational science. *Concurrency—Practice and Experience 5*, 5 (August), 425–448.

49. ○ P. Brinch Hansen 1993d. Letter to D. L. Parnas, (December 17).

50. • P. Brinch Hansen 1994a. Do hypercubes sort faster than tree machines? *Concurrency—Practice and Experience 6*, 2 (April), 143–151.

51. • P. Brinch Hansen 1994b. The programming language SuperPascal. *Software—Practice and Experience 24*, 5 (May), 467–483.

52. ⋆ P. Brinch Hansen 1994c. SuperPascal—A publication language for parallel scientific computing. *Concurrency—Practice and Experience 6*, 5 (August), 461–483. *Article 18.*

53. • P. Brinch Hansen 1994d. Interference control in SuperPascal—a block-structured parallel language. *The Computer Journal 37*, 5, 399–406.

54. • P. Brinch Hansen 1995a. *Studies in Computational Science: Parallel Programming Paradigms.* Prentice Hall, Englewood Cliffs, NJ, (March).

55. ⋆ P. Brinch Hansen 1995b. Efficient parallel recursion, *SIGPLAN Notices 30*, 12 (December), 9–16. *Article 19.*

56. • P. Brinch Hansen 1996. *The Search for Simplicity: Essays in Parallel Programming.* IEEE Computer Society Press, Los Alamitos, CA, (April).

57. • P. Brinch Hansen 1999. Java's insecure parallelism. *SIGPLAN Notices 34*, 4 (April), 38–45.

58. • P. Brinch Hansen 2000. The evolution of operating systems. In P. Brinch Hansen Ed. 2001, *Classic Operating Systems: From Batch Processing to Distributed Systems*, Springer-Verlag, New York, (January), 1–34.

59. • P. Brinch Hansen Ed. 2001. *Classic Operating Systems: From Batch Processing to Distributed Systems.* Springer-Verlag, New York, (January).

60. • C. Bron 1972. Allocation of virtual store in the THE multiprogramming system. In C. A. R. Hoare and R. H. Perrott Eds., *Operating Systems Techniques*, Academic Press, New York, 168–184.

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

62. • N. Carriero and D. Gelernter 1989. Linda in context. *Communications of the ACM 32*, 4 (April), 444–458.

63. • R. S. Cok 1991. *Parallel Programs for the Transputer.* Prentice Hall, Englewood Cliffs, NJ.

64. • M. I. Cole 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA.

65. • D. Coleman 1980. Concurrent Pascal—an appraisal. In R. M. McKeag and A. M. Macnaghten Eds., *On the Construction of Programs*, Cambridge University Press, New York, 213–227.

66. ○ Cosine Report 1971, *An Undergraduate Course on Operating Systems Principles*, P. J. Denning, J. B. Dennis, B. Lampson, A. N. Haberman, R. R. Muntz and D. Tsichritzis Eds., Commission on Education, National Academy of Engineering, Washington, DC, (June).

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

68. ○ O.-J. Dahl and K. Nygaard 1963. Preliminary presentation of the Simula language (as of May 18, 1963) and some examples of network descriptions. Norwegian Computing Center, Oslo, Norway.

69. • O.-J. Dahl and C. A. R. Hoare 1972. Hierarchical program structures. In O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Eds., *Structured Programming*, Academic Press, New York, 175–220.

70. ○ O.-J. Dahl 1993. Reviewers' comments. In P. Brinch Hansen, Monitors and Concurrent Pascal: a personal history. *SIGPLAN Notices 28*, 3 (March), 1–35.

71. ○ E. W. Dijkstra 1960. Recursive programming. *Numerische Mathematik 2*, 312–318.

72. ⋆ E. W. Dijkstra 1965. Cooperating sequential processes. Technological University, Eindhoven, The Netherlands, (September). Also in F. Genuys Ed. 1968. *Programming Languages*, Academic Press, New York, 43–112. *Article 1.*

73. ⋆ E. W. Dijkstra 1968. The structure of the THE multiprogramming system. *Communications of the ACM 11*, 5 (May), 341–346. *Article 2.*

74. ⋆ E. W. Dijkstra 1971. Hierarchical ordering of sequential processes. *Acta Informatica 1*, 2 (October), 115–138. *Article 4.*

75. • E. W. Dijkstra 1972a. Notes on structured programming, In O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Eds., *Structured Programming*, Academic Press, New York, 1–82.

76. • E. W. Dijkstra 1972b. Information streams sharing a finite buffer. *Information Processing Letters 1*, 5 (October), 179–180.

77. ⋆ E. W. Dijkstra 1975a. A synthesis emerging?, (July). In E. W. Dijkstra 1982, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York, 147–160. *Article 14.*

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

79. • E. W. Dijkstra and C. S. Scholten 1982. A class of simple communication patterns. In E. W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York, 334–337.

80. ∘ Discussions 1971. Discussions of conditional critical regions and monitors. In C. A. R. Hoare and R. H. Perrott Eds. 1972, *Operating Systems Techniques*, Proceedings of a Seminar at Queen's University, Belfast, Northern Ireland, August–September 1971. Academic Press, New York, 100–113.

81. ∘ J. A. Fellows 1993. Reviewers' comments. In P. Brinch Hansen, Monitors and Concurrent Pascal: a personal history. *SIGPLAN Notices 28*, 3 (March), 1–35.

82. ∘ J. Gosling, B. Joy and G. Steele 1996. *The Java Language Specification.* Addison-Wesley, Reading, MA.

83. • J. S. Greenfield 1991. Distributed programming with cryptography applications. *Lecture Notes in Computer Science 870*, Springer-Verlag, New York.

84. ∘ A. N. Habermann 1967. On the harmonious cooperation of abstract machines. Ph.D. thesis. Technological University, Eindhoven, The Netherlands.

85. • A. N. Habermann 1972. Synchronization of communicating processes. *Communications of the ACM 15*, 3 (March), 171–176.

86. • A. C. Hartmann 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.

87. ∘ J. M. Havender 1968. Avoiding deadlock in multitasking systems. *IBM Systems Journal 7*, 2, 74–88.

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

89. ○ C. C. Hayden 1993. Reviewers' comments. In P. Brinch Hansen, Monitors and Concurrent Pascal: a personal history. *SIGPLAN Notices 28*, 3 (March), 1–35.

90. ⋆ C. A. R. Hoare 1971. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott Eds. 1972, *Operating Systems Techniques*, Proceedings of a Seminar at Queen's University, Belfast, Northern Ireland, August–September 1971. Academic Press, New York, 61–71. *Article 5.*

91. • C. A. R. Hoare 1972. Proof of correctness of data representations. *Acta Informatica 1*, 271–281.

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

93. • C. A. R. Hoare 1973b. A structured paging system. *Computer Journal 16*, (August), 209–214.

94. ○ C. A. R. Hoare 1973c. Letter to R. M. McKeag, (October 10).

95. • C. A. R. Hoare 1974a. Hints on programming language design. In C. Bunyan Ed., *Computer Systems Reliability*, Infotech International, Berkshire, England, 505–534.

96. ⋆ C. A. R. Hoare 1974b. Monitors: an operating system structuring concept. *Communications of the ACM 17*, 10 (October), 549–557. *Article 9.*

97. ○ C. A. R. Hoare 1976a. The structure of an operating system. In *Language Hierarchies and Interfaces*, Springer-Verlag, 1976, 242–265.

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

99. ⋆ C. A. R. Hoare 1978. Communicating sequential processes. *Communications of the ACM 21*, 8 (August) 1978, 666–677. *Article 15.*

100. ○ C. A. R. Hoare 1981. A calculus of total correctness for communicating sequential processes. *The Science of Computer Programming 1*, 1–2 (October), 49–72.

101. • C. A. R. Hoare 1985. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ.

102. ○ J. J. Horning 1972. Preliminary report to Karl Karlstrom, computer science editor, Prentice-Hall, on the manuscript of "Operating System Principles" by P. Brinch Hansen, (May 1).

103. • J. H. Howard 1976. Proving monitors. *Communications of the ACM 19*, 5 (May), 273–274.

104. • M. E. C. Hull 1987. occam—a programming language for multiprocessor systems. *Computer Languages 12*, 1, 27–37.

105. ○ IBM 1965. IBM operating system/360 concepts and facilities. In S. Rosen, Ed., *Programming Systems and Languages*, McGraw-Hill, New York, 598–646.

106. ○ Inmos 1988a. *Transputer Reference Manual.* Prentice Hall, Englewood Cliffs, NJ.

107. • Inmos 1988b. *occam 2 Reference Manual*, Prentice Hall, Englewood Cliffs, NJ.

108. • G. Kahn and D. B. McQueen 1977. Coroutines and networks of parallel processes. In B. Gilchrist Ed., *Information Processing 77*, North-Holland Publishing, Ansterdam, The Netherlands, 993–998.

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

110. • J. M. Kerridge 1982. A Fortran implementation of Concurrent Pascal. *Software—Practice and Experience 12*, 1 (January), 45–55.

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

112. ◦ T. Kilburn, R. B. Payne and D. J. Howarth 1961. The Atlas supervisor. *AFIPS Computer Conference 20*, 279–294.

113. ◦ D. Knuth and J. L. McNeley 1964. SOL—A symbolic language for general-purpose systems simulation. *IEEE Transactions on Electronic Computers 13*, 8 (August), 401–408.

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

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

116. ◦ S. K. Langer 1967. *An Introduction to Symbolic Logic.* Dover Publications, New York.

117. • S. Lauesen 1975. A large semaphore based operating system. *Communications of the ACM 18*, 7 (July), 377–389.

118. ◦ D. Lea 1997. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, Reading, MA.

119. ◦ P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson and B. L. Stumpf 1983. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications 1*, 5, 842–856.

120. • B. H. Liskov 1972. The design of the Venus operating system. *Communications of the ACM 15*, 3 (March), 144–149.

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

122. ◦ R. M. McKeag 1972. A survey of system structure and synchronization techniques. Department of Computer Science, Queen's University of Belfast, Northern Ireland, (October). (Supplemented January 1973.)

123. • R. M. McKeag 1976. THE multiprogramming system. In R. M. McKeag and R. Wilson Eds., *Studies in Operating Systems*, Academic Press, New York, 145–184.

124. • R, M. McKeag 1980. A structured operating system. In J. Welsh and R. M. McKeag, *Structured System Programming*, Prentice Hall, Englewood Cliffs, NY, 229–315.

125. ◦ S. E. Madnick and J. J. Donovan 1974. *Operating Systems.* McGraw-Hill, New York.

126. • D. May 1989. The influence of VLSI technology on computer architecture. In R. Elliott and C. A. R. Hoare Eds., *Scientific Applications of Multiprocessors*, Prentice-Hall, Englewood Cliffs, NJ, 21–36.

127. ○ P. Naur Ed. 1960. Report on the algorithmic language Algol 60. *Communications of the ACM 3*, 5 (May), 299–314.

128. • P. Naur Ed. 1963a. Revised report on the algorithmic language Algol 60. *Communications of the ACM 6*, 1 (January), 1–17.

129. • P. Naur 1963b. The design of the Gier Algol compiler. *BIT 3*, 2–3, 123–140 and 145–166.

130. ○ P. Naur and B. Randell Eds. 1969. *Software Engineering.* NATO Scientific Affairs Division. Brussels, Belgium (October).

131. • P. Naur 1974. *Concise Survey of Computer Methods.* Studentlitteratur, Lund, Sweden.

132. ○ P. Naur 1975. Review of "Operating System Principles." *BIT 15*, 455–457.

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

134. ○ E. I. Organick 1972. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, MA.

135. ○ E. I. Organick 1973. *Computer System Organization: The B5700/B6700 Series.* Academic Press, New York.

136. ○ D. L. Parnas 1993. Letter to P. Brinch Hansen, November 23.

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

138. ○ G. Pólya 1957. *How to Solve It.* Doubleday, Garden City, NY.

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

140. • C. W. Reynolds 1988. The historical evolution of monitors and their languages. Department of Mathematics and Computer Science, James Madison University, Harrisonburg, VA, (August).

141. • C. W. Reynolds 1990. Signalling regions: multiprocessing in a shared memory reconsidered. *Software—Practice and Experience 20*, 4 (April), 325–356.

142. ○ C. W. Reynolds 1993. Reviewers' comments. In P. Brinch Hansen, Monitors and Concurrent Pascal: a personal history. *SIGPLAN Notices 28*, 3 (March 1993), 1–35.

143. ○ D. T. Ross 1974. In search of harmony: After-dinner talk. In W. L. van der Poel and L. A. Maarssen Eds., *Machine Oriented Higher Level Languages.* North-Holland Publishing Company, Amsterdam, The Netherlands, 445–447.

144. • O. Roubine and J.-C. Heliard 1980. Parallel processing in Ada. In R. M. McKeag and A. M. Macnaghten Eds., *On the Construction of Programs*, Cambridge University Press, New York, 193–212.

145. ○ A. C. Shaw 1974. *The Logical Design of Operating Systems.* Prentice-Hall, Englewood Cliffs, NJ.

146. ○ S. K. Shrivastava and F. Panzieri 1982. The design of a reliable remote procedure call mechanism. *IEEE Transactions on Computers 31*, 7 (July), 692–697.

147. • A. Silberschatz, J. Peterson and P. Galvin 1992. *Operating System Concepts* (third edition). Addison-Wesley Publishing, Reading, MA, 1992.

148. • C. R. Snow 1992. *Concurrent Programming*. Cambridge University Press, New York.

149. • F. Stepczyk and D. Heimbigner 1979. Application of a concurrent programming language. In P. Wegner Ed., *Research Directions in Software Technology*, MIT Press, Cambridge, MA, 666–671.

150. • A. S. Tanenbaum and R. van Renesse 1988. A critique of the remote procedure call mechanism. In R. Speth Ed., *Research into Networks and Distributed Applications*, Elsevier Science Publishers, Amsterdam, The Netherlands, 775–782.

151. ○ B. H. Tay and A. L. Ananda 1990. A survey of remote procedure calls. *Operating Systems Review 24*, 3 (July), 68–79.

152. ○ A. Trew and G. Wilson Eds. 1991. *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*. Springer-Verlag, New York.

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

154. • R. L. Wexelblat Ed. 1981. *History of Programming Languages*. Academic Press, New York.

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

156. ○ N. Wirth 1969. Letter to P. Brinch Hansen, (July 14).

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

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

159. • N. Wirth 1977b. The use of Modula. *Software—Practice and Experience 7*, 1 (January–February), 37–65.

160. • N. Wirth 1977c. Design and implementation of Modula. *Software—Practice and Experience 7*, 1 (January–February), 67–84.

161. ○ W. A. Wulf, E. S. Cohen, W. M. Corwin, A. K. Jones, R. Levin, C. Pierson, and F. J. Pollack 1974. Hydra: the kernel of a multiprocessor operating system. *Communications of the ACM 17*, 6 (June), 337–345.