A Keynote Address on Concurrent Programming*

(1979)

Delivered at COMPSAC '78, this address draws a parallel between the major development phases of the first 20 years of concurrent programming and the present challenge of distributed computing.

Introduction

This keynote address summarizes the highlights of the first 20 years of concurrent programming (1960–80) and takes a look at the next 20 years (1980–2000).

A concurrent program is one that enables a computer to do many things simultaneously. Concurrent programming is used to increase computer efficiency and cope with environments in which many things need attention at the same time. Although there are good economic and conceptual reasons for being interested in concurrent programs there are major difficulties in making these programs reliable.

The slightest programming mistake can make a concurrent program behave in an irreproducible, erratic manner that makes program testing impossible. The following describes how this problem was gradually solved by software engineers and computer scientists. This development is seen as an initial hardware challenge followed by a software crisis, a conceptual innovation, and language development which in turn led to formal understanding and hardware refinement. The paper draws a parallel between this evolution of ideas and the present challenge of distributed computing.

*P. Brinch Hansen, A keynote address on concurrent programming. Keynote address for the IEEE Computer Software & Applications Conference, Chicago, IL, November 1978. *Computer 12*, 5 (May 1979), 50–56.

The Development Cycle

When you look at concurrent programming on a time-scale of decades you will see that it went through several stages of development, each lasting about 5 years:

Hardware challenge	(1955–60)
Software crisis	(1960–65)
Conceptual innovation	(1965–70)
Language development	(1970–75)
Formal understanding	(1975–)
Hardware refinement	(1980–)

At the beginning of this period new hardware developments make concurrent programming both possible and essential. As programmers experiment with this new idea they are gradually led to the development of extremely complicated systems without much of a conceptual basis to rely on. Not too surprisingly these systems soon become so unreliable that the phrase "software crisis" is coined by their designers. By then the importance of the problem is recognized by computer scientists, who start a seach for abstract concepts that will simplify the understanding of concurrent programs. Once the essence of the problem is understood a notation is invented for the basic concepts, and it now becomes possible to define them so precisely that they can be incorporated into new programming languages. This language notation in turn enables theoreticians to develop a more formal understanding of the problem. At the same time, the new language concepts inspire innovative computer designers.

At this point (if not sooner) new hardware possibilities start another development cycle. One must indeed agree with Alan Perlis that "hardware drives the field," but one must also add that abstractions make it manageable.

We will look at each of the stages that concurrent programming went through and see what the next challenge is likely to be.

The Hardware Challenge

Around 1955 computer architecture changed drastically with the invention of *large magnetic core stores* and *asynchronously operating peripheral devices.* It now became possible to write large programs of 10,000–1,000,000 machine

instructions. At the same time interrupts made it possible to write concurrent programs that could switch a fast processor among its much slower peripheral devices and make them operate simultaneously.

The intellectual challenge of this technological revolution was formidable. For the first time programs became too large to be understood completely by a single programmer. In response to this challenge computer programmers invented the first abstract programming languages, Fortran and Algol 60, and made their compilers some of the best understood and most reliable system programs we know. All this happened in less than 10 years—a most impressive achievement (Wexelblat 1978).

The capabilities for simultaneous execution of several tasks on one computer did, however, create a serious problem that took much longer to solve: Programming errors could now cause a concurrent program to behave in an erratic, time-dependent manner. These errors were extremely difficult to find since their effect varied from one execution to the next even when the input data remained the same. It has taken 20 years to cope with this problem of concurrency.

If you look at computers from a programmer's point of view the main problem is to master the complexity of the hardware innovations that were introduced two decades ago. By comparison mini- and microcomputers are not revolutionary at all. Their economic impact and the numerous possibilities for new applications are far reaching. But they have not, so far, posed new programming problems of the same difficulty (thank heaven).

The Software Crisis

The slowness of peripheral devices made asynchronous operation essential for efficient computer operation. But the pitfalls of concurrency made it equally important to present the user with a simple, sequential interface to the machine. The new system programs that were supposed to make a concurrent computer system simple, reliable, and efficient were called *operating systems*.

Some of the early batch processing systems, such as Atlas (1961) and Exec II (1962), were both efficient and simple. But they were not entirely reliable. In looking back Bill Lynch (1972) observed that "several problems remained unsolved with the Exec II operating system and had to be avoided by one *ad hoc* means or another. The problem of deadlocks was not at all understood in 1962 when the system was designed. As a result several

annoying deadlocks were programmed into the system."

The early timesharing systems, such as CTSS (1962) and SDC Q-32 (1964) were also of modest size.

Now, when faced with a new idea, programmers have an irresistable urge to push it to its natural limits and then beyond. The operating systems of the next generation were complex beyond human comprehension. The Multics system (1965) required 200 man-years of development effort, and OS360 (1966) a staggering 5000 man-years. Because of its size OS360 became quite unreliable. In 1969 Hopkins said this: "We face a fantastic problem in big systems. For instance, in OS360 we have about 1000 errors in each release and this number seems to be reasonably constant" (Naur 1969).

At this point it had become common for large operating systems to fail daily, and it was doubtful whether they were achieving their original aim of ensuring efficient, reliable computer operation. There was a clear feeling at this point that it was just not possible to design these large programs without some conceptual basis that would make them more understandable.

The importance (and failure) of operating systems had by now become clear to computer scientists who, like all other computer users, were forced to depend on these systems in their own computing centers. And so the search for abstractions began.

The Conceptual Innovation

Seen in retrospect this development was clearly a search for concepts that would make it possible to divide a concurrent program into smaller *asynchronous modules* with *time-independent behavior*.

The idea of dividing a concurrent program into *sequential processes* that are executed asynchronously was by far the most important innovation. This idea and its implementation were pioneered at MIT in the CTSS project (Saltzer 1966).

A process is a program module that consists of a data structure and a sequence of statements that operates on it. If each process only operates on its own data then it will behave in a completely predictable manner each time it is executed with the same data. Hardware protection mechanisms can prevent processes from referring to each other's data structures by mistake.

It now became possible to perform unrelated tasks simultaneously without time-dependent interference. However, if processes share computer resources or cooperate on common tasks then they must also be able to share

data in a controlled manner. During the late sixties the main focus was the invention of safe methods for synchronizing processes which share data.

Dijkstra's THE system (1968a, 1968b, 1971) is the milestone of this era. It introduced most of the concepts on which our present understanding of concurrent programming rests. Dijkstra noticed that all communication among processes boils down to performing operations on common data. But if several processes operate simultaneously on the same variables at unpredictable speeds, the result will be unpredictable since none of the processes have any way of knowing what the others are doing to the variables. Dijkstra therefore concluded that it is essential to perform the operations on the common variables strictly one at a time. If one process is operating on common variables then the machine must delay further operations on the same variables until the present operation is finished. Dijkstra introduced the name *critical region* for operations on common variables which take place one at a time.

Critical regions only prevent competing processes from using common variables simultaneously. But they do not help in transmitting data correctly from one process to another. In looking at the problem of process communication, Dijkstra began by studying the simplest possible case in which timing signals are sent from one process to another. For this purpose he invented a data type, called a *semaphore*.

A signal operation permits a process to transmit a timing signal through a semaphore variable to another process which receives the signal by performing a wait operation. In a concurrent system, the programmer cannot predict the relative speeds of asynchronous processes. It is therefore impossible to know whether one process will try to send a signal before another is ready to receive it (or vice versa). Dijkstra removed this problem by defining the semaphore operations in such a way that it doesn't matter in which order they are initiated. If a process tries to receive a timing signal before it is available, the wait operation will simply delay the process until another process sends the next signal. Conversely, if signals temporarily are being sent faster than they can be received, they will simply be stored in the semaphore variable until they are needed.

The *commutativity* of semaphore operations made process synchronization time-independent. Dijkstra then went on to show how critical regions and message buffers can be implemented by means of semaphores.

Dijkstra's multiprogramming system also illustrated the conceptual clarity of *hierarchical structure*. His system consisted of several program lay-

ers which gradually transform the physical machine into a more pleasant abstract machine that simulates several processes which share a large, homogeneous store and several virtual devices. These program layers can be designed and studied one at a time.

His co-worker Habermann (1967) showed that a hierarchical ordering of resource requests and message communication also can prevent deadlocks.

Around 1970 researchers began to invent language notation for these powerful new concepts.

Language Development

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 language notation for it. But in practice this is hard to do. The main problem is to replace an intuitive, vague idea with a precise, unambiguous definition of its meaning and restrictions. The mathematician Polya (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 very often. Otherwise, it will just increase the complexity of the language at no apparent gain. The meaning and rules of a programming language 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. The 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 notation that suppresses irrelevant detail immediately becomes obvious. So, although Dijkstra's THE system was implemented in assembly language, he found it helpful to invent a language notation for concurrent processes in his description (Dijkstra 1968a).

The following example of Dijkstra's *concurrent statement* shows two sequential statements that are executed simultaneously:

```
var this, next: line
cobegin consume(this); input(next) coend
```

While one statement is consuming a line of text, called *this*, another statement is inputting the *next* line. The concurrent statement terminates when all the component statements are terminated.

Hoare (1972a) pointed out that the concurrent statement only has a predictable effect if the statements within it operate on different variables. In this example, the consumer and the input statements refer to different variables (this and next). If the programmer by mistake lets both statements refer to the same variable, the effect of the concurrent statement will be time-dependent.

To prevent time-dependent programming errors a compiler should be able to recognize the *private variables* of a process and make them inaccessible to other processes. Unfortunately, this is difficult to do in more complicated examples involving procedures and global variables. The solution to this problem will be described later.

Although it is essential to make some variables accessible to one process only, it is also necessary to enable processes to share other variables to make cooperation and communication possible.

In 1971–72 notations were proposed for associating a *shared variable* with the *critical regions* that operate on it (Hoare 1972a; Brinch Hansen 1972). A shared integer used as a clock is a good example:

```
var clock: shared integer
```

Processes can either increment or read this clock by statements of the form:

```
tick:     region clock do
          clock := (clock + 1) mod max

read(x):  region clock do x := clock
```

The compiler checks that a shared variable is accessed only within critical regions. The computer guarantees that these regions are executed one at a time without overlapping.

Hoare also invented the beautiful concept of a *conditional critical region* which is delayed until a shared variable satisfies some condition (defined by a boolean expression). A good example is a message buffer consisting of a single line *slot* and a boolean indicating whether or not it is *full*:

```
var buffer: shared record
                slot: line
                full: boolean
            end
```

The send operation is a conditional critical region that is executed when the buffer is empty:

```
send(m):  region buffer when not full do
          begin slot := m; full := true end
```

The receive operation is similar:

```
receive(m):  region buffer when full do
             begin m := slot; full := false end
```

At that time it did not seem possible to implement conditional critical regions efficiently on a single processor. The problem was to limit the repeated evaluation of boolean expressions until they become true. As a compromise between elegance and efficiency process *queues* (also called "events" or "conditions") associated with shared variable were proposed (Brinch Hansen 1972).

At that time Dijkstra (1971) suggested that the meaning of process interactions could be further clarified by combining all operations on a shared data structure into a single program module (instead of scattering them throughout the program text).

In 1973 a language notation for this *monitor* concept was proposed (Brinch Hansen 1973). The data representation of a message buffer together with the send and receive operations on it now looked like this:

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

procedure send(m: line)
when not full do
begin slot := m; full := true end

procedure receive(var m: line)
when full do
begin m := slot; full := false end

begin full := false end
```

The monitor includes an initial statement that makes the buffer empty to begin with. In a later paper Hoare (1974) also described the monitor concept and illustrated it with examples.

A central theme in this development was an attempt to replace earlier hardware protection mechanisms by compilation checks. The monitor concept enables a compiler to check that send and receive are the only operations performed on a message buffer. Once the buffer monitor has been tested systematically the compiler prevents other program modules from using it incorrectly. This tends to localize errors in new, untested modules and prevent them from causing obscure effects in old, tested modules.

The elimination of execution checks was not done just to make compiled programs more efficient. In program engineering, compilation and execution checks play the same roles as preventive maintenance and flight recorders do in aviation. The latter only tell you why a system crashed; the former prevents it. This distinction is essential in real-time systems that control vital functions in society. Such systems must be highly reliable *before* they are put into operation.

The monitor concept solved the problem of *controlled access* to shared variables. The earlier problem of controlling the access to private variables was solved by declaring each process and its local variables as a separate program module:

```
process producer
var next: line
cycle input(next); buffer.send(next) end

process consumer
var this: line
cycle buffer.receive(this); consume(this) end
```

This language notation makes it obvious to the program reader and the compiler that the variable *next* only can be used within the producer process.

The first programming language based on processes and monitors was *Concurrent Pascal*. It was defined and implemented in 1974 (Brinch Hansen 1975). By the end of 1975 Concurrent Pascal had been used to write three minicomputer operating systems of 600–1400 lines each. The development and documentation effort of each system only took a few weeks (Brinch Hansen 1976, 1977). A later language, *Modula* (Wirth 1977), is also based on the process and monitor concepts.

These language concepts had a dramatic impact on the structure of concurrent programs. It now became natural to build a concurrent program out of modules of one page each. Since each module defines all the meaningful operations on a single data structure (private or shared), the modules can be studied and tested one at a time. As a result these concurrent programs became more reliable than the hardware they ran on. And their simplicity made it possible to publish the entire text of a concurrent program of 1300 lines (Brinch Hansen 1976).

It is interesting that sequential programmers independently were led to the discovery of program modules which combine data repesentations and procedures into units (Hoare 1972b). But although the two developments led to the same conclusions the motivations were different: concurrent programmers were gradually led to modularity simply by their desire to master synchronization and prevent race conditions. These problems do not occur in sequential programs. Sequential programmers were motivated by more abstract concerns for clarity and the desire to make program verification simpler.

Formal Understanding

Once you have a notation for a concept it becomes possible to refine it further and get a more formal understanding of its properties. The impact of notation on discovery has been expressed very well by Susanne Langer (1967):

"There is something uncanny about the power of 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."

It is no coincidence therefore that the development of language notation for concurrent programming immediately inspired theoretical work on program verification. Hoare's first paper on concurrent programming (1972a) contains axiomatic definitions of the meaning of concurrent statements and critical regions. A later paper by Hoare (1974) defines the effect of queue manipulation within monitors. The development of verification rules for concurrent programs with conditional critical regions was carried further by Owicki and Gries (1976).

It remains to be seen what effect these theories will have on language refinement and program reliability. Most researchers would agree that our theoretical understanding of concurrency is still in its infancy. A successful approach in this area will almost certainly require that computer scientists go beyond well-understood exercises and concern ourselves with model systems of a non-trivial size.

Hardware Refinement

The trend of decreasing hardware costs and increasing software costs is likely to continue due to better production methods and continued inflation. At the moment the use of abstract programming languages is the only effective way of reducing software costs. Unfortunately, present computer architectures do not support abstract languages efficiently compared to machine language. A real-time programmer is therefore faced with a meaningless choice among cost, reliability, and efficiency. The solution is quite obvious: we must build computer architectures that support our programming concepts directly.

A few years after the invention of the block and procedure concepts of Algol 60 the first stack computers appeared. It did, however, take more than a decade for this idea to be generally adopted by most computer manufacturers.

A similar development is now taking place in concurrent programming. The microprocessor technology makes it possible to build computer architectures that will support the process and monitor concepts directly. A recent proposal envisions a computer with 10 microprocessors. Each processor has a local store dedicated to a single process. The processors share a common store that contains the monitors. The computer has no interrupts and does not multiplex its processors among several processes (Brinch Hansen 1978b).

I would expect an increasing number of computer architectures to be oriented towards the support of concurrent programming languages for real-

time applications.

For applications that are of interest to a large number of people it will be economical to specialize the hardware even further. In those cases it seems very attractive to write a concurrent program in an abstract language that hides machine detail, test it on an existing machine, and then derive the most straightforward specialized architecture from the program itself.

Like the development of our theoretical understanding, the design of new computer architectures for concurrent programming has started and will probably continue for another decade.

The Next Challenge: Computer Networks

It has taken 20 years to design reliable computer systems in which concurrent processes share storage. And now hardware technology has provided another challenge: microcomputer networks in which processors communicate by input/output only (without any common storage). This seems a natural approach to real-time applications in which geographically distributed functions must be coordinated.

Anyone who took the word "abstraction" to mean "machine-independent" suddenly discovered that abstract programming languages merely hide the irrelevant differences between similar computer architectures. The procedure concept is still fundamentally tied to the existence of a common store for parameter passing. And the people who developed monitors for concurrent programming also took this assumption for granted.

Now it may seem that the solution to the distributed processing problem is simple: message passing between processors connected by cables is all that is needed. And message passing (one of the oldest ideas in concurrent programming) we surely understand very well. Unfortunately, it is not that easy.

What we do understand is *deterministic message passing* in which a receiving process waits until another process sends a message on a given line. In such a system each process performs a completely predictable transformation of its input to its output. The analysis of individual processes must be supplemented with a global analysis of termination (or absence of deadlocks). This can be guaranteed by a hierarchical ordering of processes into "masters" and "servants."

A recent paper by Hoare (1978), however, makes it clear that one must also include *nondeterministic message passing*—a far more complex problem.

An obvious example is a process that functions as a buffer between two other processes. The buffer process cannot predict whether its environment will ask it to receive or send a message next. Consequently, it cannot commit itself to waiting until it receives a message on the input line, for this would make it unable to respond to a request for sending a message on the output line. Conversely, it cannot commit itself to wait until it is asked to send a message either, for this would make it unable to receiver further messages in the meantime, thereby slowing down the producer process unnecessarily.

What is needed therefore is the ability of a process to delay itself until it receives either a request for sending or receiving. It must then be able to perform one of two actions depending on what it was asked to do.

The problem is further complicated by the finite storage capacity of a buffer process. When the buffer is full, the process cannot accept further input; and when it is empty, the process cannot deliver further output. Hoare is therefore led to introducing a non-deterministic statement of the form:

```
when
  not full(buffer) & input(x): put(x, buffer)
  not empty(buffer) & output(x): get(x, buffer)
end
```

These *communicating sequential processes* seem somewhat inconvenient for the programming of processes that schedule other processes. To handle this problem the concept of *distributed processes* has been proposed (Brinch Hansen 1978a). It combines the process and monitor concepts and enables one process to call a procedure within another process when the latter process is waiting for some condition to be satisfied by its own variables. The parameter passing between processes can be done by a single input operation before a process interaction followed by a single output operation afterwards.

The practicality of these recent proposals has not yet been established. They have not even been implemented and are not understood formally. Their main value is to make it clear that distributed computing will require new concepts.

If the history of concurrent programming is about to repeat itself we should expect the new hardware challenge to lead to a software crisis as the technology is being used in real-time applications by means of *ad hoc* programming techniques. The search for concepts, languages, and theory will then start again. This will take longer than we may think. I would expect distributed computing to be reasonably well understood by the year 2000.

Acknowledgements

This work was supported by the Office of Naval Research under contract number NR049-415.

References

Brinch Hansen, P. 1972. Structured multiprogramming. *Communications of the ACM 15*, 7 (July), 574–578. *Article 4.*

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

Brinch Hansen, P. 1975. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering 1*, 2 (June), 199–207. *Article 7.*

Brinch Hansen, P. 1976. The Solo operating system. *Software—Practice and Experience 6*, 2 (April–June), 141–205. *Articles 8–9.*

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

Brinch Hansen, P. 1978a. Distributed Processes—A concurrent programming concept. *Communications of the ACM 21*, 11 (November), 934–941. *Article 14.*

Brinch Hansen, P. 1978b. Multiprocessor architectures for concurrent programs. *Proceedings of the ACM 78 Conference*, Washington, DC, (December), 317–323.

Dijkstra, E.W. 1968a. Cooperating sequential processes. In *Programming Languages*, F. Genuys, Ed., Academic Press, New York, 43–112.

Dijkstra, E.W. 1968b. The structure of the "THE"-multiprogramming system. *Communications of the ACM 11*, 5 (May), 341–346.

Dijkstra, E.W. 1971. Hierarchical ordering of sequential processes. *Acta Informatica 1*, 115–138.

Habermann, A.N. 1967. On the harmonious cooperation of abstract machines. Technological University, Eindhoven, The Netherlands.

Hoare, C.A.R. 1972a. Towards a theory of parallel programming. In *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrott, Eds., Academic Press, New York, 61–71.

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

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

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

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

Lynch, W.C. 1972. An operating system design for the computer utility environment. In *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrott, Eds., Academic Press, New York, 341–350.

Naur. P, and Randell, B., Eds. 1969. *Software Engineering.* NATO Scientific Affairs Division, Brussels, Belgium, (October), 20.

Owicki, S., and Gries, D. 1976. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM 19*, 5 (May), 279–288.

Polya, G. 1957. *How to Solve It.* Doubleday, Garden City, NY.

Saltzer, J.H. 1966. Traffic control in a multiplexed computer system. Massachusetts Institute of Technology, Cambridge, MA, (July).

Wexelblat, R.L., Ed. 1978. *ACM Conference on the History of Programming Languages*, (Preprints), Los Angeles, CA, (June). In *SIGPLAN Notices 13*, 8 (August).

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