5

SHAPING A NEW FIELD 1970–72

Alan Perlis invites me to spend a year at Carnegie-Mellon – Emigration to America – Niklaus Wirth defines Pascal – Driving home in a blizzard – Discussing the future of concurrent programming in Marktoberdorf and Belfast – Alan Perlis tells stories – Mad King Ludwig and Thomas Edison – The first modern book on Operating System Principles.

Alan Perlis was an early pioneer in the development of programming languages and compilers. He was involved in the definition of the programming languages Algol 58 and Algol 60. In a retrospective talk, he said (Perlis 1981):

> One of the things we learned about computing in the 1950s was that there are no bounds to the subject. It cannot be put into a tidy receptacle. Everywhere that computing has been embedded in some other discipline, it has not flowered. Computing is not part of electrical engineering; it is not part of mathematics; it is not part of industrial administration. Computing belongs to itself. The reason this *is* the computer age is precisely because of that.

In 1962, Perlis was forty years old and director of the computation center at Carnegie Institute of Technology in Pittsburgh. He believed that "the programming and using of computers deserve an early appearance in the university curriculum for the educated man." At an MIT symposium on "Computers and the World of the Future," he added (Perlis 1962):

> Perhaps I may have been misunderstood as to the purpose of my proposed first course in programming. It is not to teach people how to program a specific computer, nor is it to teach some new

> language. The purpose of a course in programming is to teach people how to construct and analyze processes. I know of no other course that the student gets in his first year in a university which has this as its sole purpose.

Today, when every department of computing and engineering offers programming courses, it may be difficult to appreciate just how farsighted Perlis was—until you realize that he took it for granted that every student of the arts and sciences eventually would be required to take such a course. Forty years later, that still hasn't happened.

In 1965, Carnegie Tech became one of the first universities to create a graduate department of computer science. Alan Perlis, Allen Newell, and Herbert Simon were the driving forces behind the establishment of computer science as an independent discipline at Carnegie.

The appointment of the visionary MIT professor, Joseph Carl Robnett Licklider, as director of information processing research at ARPA (The Advanced Research Projects Agency at the US Department of Defense) was a major piece of luck for Carnegie Tech. Licklider selected MIT and Carnegie as the first two ARPA centers of excellence. While MIT would develop timesharing, Carnegie would have complete freedom to explore the fundamentals of computing. I cannot think of any other country and research agency that would distribute government funds in such an informal way.

Of course, money by itself does not explain how Carnegie was able to establish one of the finest schools of computer science in the world (Perlis 1981):

> I think computers have flowered in this country because of our national style of accomplishing things. This country has always supported enterpreneurial activities—people who have ideas and are willing to sweat to bring them out. Computers flower in such an environment...The Soviet Union, having a large centralized society, needs computers much worse than we do. Yet they are totally unable to produce them, totally unable to apply them in anywhere near the profusion that we find here. They have the wrong kind of society for the instrument they most badly need.

Perlis was the founding editor of *Communications of the ACM*, a past president of the Association for Computing Machinery, and now the first chair of computer science at Carnegie. His administrative tasks cannot have

left him much time for concentrated intellectual work. Apart from Peter Naur's Algol 60 report I find few references to his work in textbooks.

Alan was completely bald and had no eyebrows. He looked (and was) extremely intelligent. He was immensely charming with an endless supply of anecdotes and jokes. And highly original in his thinking. They said that if a student met him in the elevator, Al would propose several thesis topics on the spot. The trouble was that only one of them was worth pursuing and the student had no idea which one that might be.

His faculty and students loved him. At parties they would gather around him and listen to his anecdotes and words of wisdom. He was famous for his wise and amusing sayings (Perlis 1982):

- Syntactic sugar causes cancer of the semicolon.
- In the long run every program becomes rococo—then rubble.
- Simplicity does not precede complexity, but follows it.
- Structured Programming supports the law of the excluded muddle.
- A LISP programmer knows the value of everything, but the cost of nothing.
- Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.
- When we write programs that "learn," it turns out we do and they don't.
- Because of its vitality, the computing field is always in desperate need of new cliches: Banality soothes our nerves.
- Editing is a rewording activity.

He had sound advice on how to deal with university administration (Perlis 1981):

> My attitude has always been that, if you are right, the administrators will accede to your wishes. I haven't been disappointed. By and large, administrators are always looking for people to tell them what they ought to be doing, rather than being confronted with a decision that they have to make on which they have no

> information with which to make that decision, so the natural technique is to postpone or form another committee. Instead, one really ought to go to them and say, "Do this because it's right."

Alan Perlis reminded me of Niels Ivar Bech. Both of them were visionary thinkers and brilliant leaders, who were ahead of their time in recognizing the need for research and education in computing. Like Bech at Regnecentralen, Perlis's mission in life was to create a place where creative people could flourish.

In an interview after Perlis's death, in 1990, Allen Newell said: "People would go to conferences and listen to every word he said, because every time he talked about a topic, he was absolutely right about the way it should be. That's why he was the first Turing Award winner. Not at all for any technical contribution, really, but because he epitomized the nature of computer science."

⋆ ⋆ ⋆

Dijkstra's first doctoral student, Arie Nicolaas Habermann, was a former math teacher in the Netherlands. In 1968, after receiving his PhD, Nico joined the computer science department at Carnegie-Mellon University (as it was now called).

At the Technological University of Eindhoven (THE) in the Netherlands, he had participated in the implementation of Dijkstra's famous THE multiprogramming system. This system consisted of several "program layers," which gradually transformed the physical machine into an ideal abstract machine for multiprogramming. The program layers could be designed and tested one at a time.

At a time, when the term "software crisis" was coined to describe the sad state of operating systems, Dijkstra (1968b) made a 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.

Dijkstra proved that the THE system was deadlock-free. (A deadlock is a situation in which concurrent processes wait indefinitely for events that will never occur.) In his thesis, Habermann extended Dijkstra's proof to a layered system with any number of levels.

I first met Nico at the NordSAM conference in Stockholm, Sweden, in August 1964. Over the years, we kept in contact. Like his thesis advisor, Edsger Dijkstra, he was well aware of the RC 4000 multiprogramming system. On January 2, 1969, he invited me to visit Perlis's department at Carnegie-Mellon:

> It is my pleasure to invite you officially to spend some time during the spring in Pittsburgh and to visit our department. We would like you to spend about two weeks with us, during which we expect you to tell us about your work.

I arrived in Pittsburgh in March 1970 and stayed with Nico, his wife Marta, and their children, Eveline, Irene, Marianne, and Frits. Every morning, Nico and I drove in his Volkswagen from the suburb of Mount Lebanon to the university in Schenley Park.

In a classroom at Carnegie, I lectured on the RC 4000 multiprogramming system. The audience of about 30 people included Alan Perlis, Gordon Bell, Bill Wulf, and, of course, Nico. Among the graduate students, I remember Anita Jones and Rudy Krutar.

During my visit, I mentioned that I was planning to spend a year abroad writing a textbook on operating systems. Perlis immediately said: "I am as greedy as the next man—why don't you write it at Carnegie?" Since Nico was also planning to write such a book, we agreed to make it a joint effort.

A month later Perlis sent me an official invitation to spend a year in Pittsburgh (Fig. 5.1). As you can tell, he could charm your socks off! This letter would change my life dramatically.

⋆ ⋆ ⋆

On November 1, 1970, Milena and I left Denmark with our children, Mette (age 4) and Thomas (age 3), and traveled to Pittsburgh on a Lufthansa 707 jetplane. In Mount Lebanon, we rented a small house on 603 Oxford Boulevard. Our rented furniture included a sofa stuffed with sawdust that slowly accumulated on the floor. Upstairs, we slept on mattresses on the floor.

Carnegie-Mellon University
Department of Computer Science
Schenley Park
Pittsburgh, Pennsylvania 15213

April 17, 1970

Civ.ing. Per Brinch Hansen
Hostrups Have 32/6
1954 Copenhagen V
Denmark

Dear Per:

With this letter I should like to formally invite you to spend either 10 months or 12 months as a research associate in the Department of Computer Science. It is my understanding that you would like to commence on or about October 1, 1970, and this is perfectly satisfactory with the department.

The salary will be $15,000 for 10 months or $18,000 for 12 months. Furthermore, the department agrees to pay 1/2 of your reasonable traveling expenses from Copenhagen to Pittsburgh and return.

We are pleased to make this offer since the talent and experience you would provide would be of great value to the research program of this department. Furthermore, the department is pleased that a person of your exceptional ability will be collaborating with members of our faculty in important research to further our understanding of the nature of complex computer operating systems.

Please keep me informed of the progress in your application for visa so that we may provide whatever assistance is needed from this end.

Needless to say the entire faculty is looking forward to having you with us this coming school year.

Sincerely yours,

Dr. Alan J. Perlis, Head
Department of Computer Science

Figure 5.1 Invitation from Alan Perlis.

From Marcus Motor downtown, I bought a new Volvo for $3,400. In Denmark, where sales taxes doubled the price of a car, this was considered a large car. But, in America our neighbors referred to it as "your cute little car" and Perlis offered the opinion that "it rides like a truck." It turned out to be a lemon that stalled if you tried to turn left at a traffic signal. To avoid this in the middle of heavy traffic, I would drive around the block turning right three times instead.

In Denmark the kids attended a nursery school that did not teach religion. When I asked Perlis, who was Jewish, for advice on how to chose a similar one in Pittsburgh, he suggested putting them in a Jewish nursery school. We took his advice and it worked out well.

It was a lonely time for Milena. Her H-4 visa did not permit her to work in America. When I was at work and the children at nursery school, she was alone in the house without a car.

⋆ ⋆ ⋆

After the first visit to America, I began outlining my book and collect papers on operating systems from Regnecentralen's excellent library. As soon as I started writing, it became clear that I needed a programming language to express operating system functions concisely without unnecessary trivia. I began using Pascal, which had a much richer set of data structures than Algol 60. Niklaus Wirth had just published the Pascal report, while three of his coworkers completed the first compiler (for the CDC 6600 computer). In hindsight, my choice of Pascal was an inspired gamble. I had no way of knowing whether this new language would ever be widely available and used for teaching!

The idea of writing the book jointly with Nico Habermann turned out to be a grave mistake. At Regnecentralen, I was used to an atmosphere of frank, critical discussion. If I thought Nico's ideas were problematic, I would tell him so. Habermann was equally critical of my approach. He hated Pascal with a passion and would eventually write a scathing criticism of the language, which the editor of Acta Informatica asked him to tone down (Habermann 1973). He had a point—in its lack of conciseness, the Pascal report was a step backwards compared to Peter Naur's brilliant Algol 60 report. Nevertheless, the language features of Pascal were (in my opinion) a significant contribution to programming language design.

Coming from the Netherlands, Nico must have felt under great pressure to live up to the expectations of a leading department in a foreign country.

Throughout his academic career, he would remain true to the high standards of software design he had learned from Dijkstra. However, as a PhD student he naturally stood in Dijkstra's shadow. When he joined Carnegie as a visiting researcher, Nico was 36 years old. His challenge was to start independent research at an age when most research careers already have peaked.

I, on the other hand, had already had some success with the RC 4000 project. Whereas Habermann had to teach and advise students, I had all the time in the world to work on my book. I made rapid progress, and he did not. Eventually, the pressure became too much for him, and one day, he left my house in a huff. The next day, I was extremely relieved to find that Perlis reacted calmly to the news of our breakup: "We are going to need many books on operating systems. Nico and you should write two separate books on the subject."

⋆ ⋆ ⋆

I admired the way American computer scientists kept themselves informed about ongoing research at other universities and research centers. When I started working at Carnegie-Mellon, I was immediately invited to give seminars at leading universities and research centers:

Toronto (November 30, 1970).
Princeton (December 16, 1970).
IBM Research Center (March 1, 1971).
Cornell (April 12, 1971).
MIT (May 17, 1971).
Berkeley (October 13, 1971).
Michigan (December 13, 1971).

These talks not only gave me a chance to meet leading computer scientists, they were also an opportunity for Milena and me to drive with our children around the eastern part of North America.

In Toronto I met members of the Sue operating system group: Rick Holt, Jim Horning, and Dennis Tsichritzis. They were building an operating system, named Sue, for the IBM 360 computers using Dijkstra's layered approach.

This was our first visit to Niagara Falls. "Niagara" is an American Indian word meaning "thundering waters." Every second, 350,000 gallons of water flow over the 2,300 feet wide falls and crash some 180 feet below. During the

day, it is truly one of the natural wonders of the world. I was less enchanted by the view at night, when the falls are illuminated by floodlights in pink, green and white.

One evening we were driving home from a visit (I don't remember which one) to Boston. As we reached Interstate 80 in Pennsylvania, it started snowing heavily as dark fell. This highway runs east-west in the Allegheny Mountains. I should have noticed that there was a reason why truck drivers were leaving the highway, one by one. Although we were now alone on this snowy road, I continued stubbornly. At one point, the snow fell so heavily that I had to roll down the front window and wipe the windshield manually while I was driving. After driving 600 miles, we reached Pittsburgh early in the morning. Later, I read that people froze to death every year when their cars got stuck in the snowy mountains.

⋆ ⋆ ⋆

In the summer of 1971, I attended two scientific meetings in Europe that would have a deep influence on my work. The first one was the International Summer School on Program Structures and Fundamental Concepts of Programming, July 19–30. It was organized by Professor Fritz Bauer, Technical University of Munich. The meeting was held in Marktoberdorf, a small town in Southern Germany. I was one of seven invited speakers:

- Rudy Bayer, On the Structure of Data and Application Problems.
- Per Brinch Hansen, Operating System Principles.
- Edsger Dijkstra, Hierarchical Organization of Sequential Processes.
- Tony Hoare, Proof of Programs.
- Alan Perlis, Conversational Languages.
- John Reynolds, Lattice Theoretic Models for Programming Language Definition.
- Niklaus Wirth, The Representation, Implementation and Application of Fundamental Concepts of Programming.

Over the course of the Summer School, each of us gave about ten lectures. The 100 (or so) participants had submitted applications to attend and were

selected by the directors of the Summer School. The speakers attended every lecture and participated in the lively discussion. After dinner in the local Hotel Sepp, the lecturers and students would continue the discussion over a glass of beer.

In my recollections of the Summer School I have included some of Dijkstra's (1971b) impressions of the speakers (translated from Dutch).

With deep insight, Dijkstra (1971a) explained his layered approach to operating system design in greater detail. He also demonstrated how to prove the correctness of fundamental algorithms with semaphores. He introduced and solved the scheduling problem of the "dining philosophers," which poses subtle dangers of deadlock and unfairness —described in flamboyant terminology as "deadly embrace" and "starvation." (A scheduling policy is "unfair" if it consistently favors some processes over others.)

In his lectures, Hoare presented his method for formal verification of sequential programs. This was still unfamiliar territory to me. I remember Tony asking me how I would prove the correctness of concurrent programs with message communication. I had no idea what he meant by the question.

> Among the speakers, C. A. R. Hoare undoubtedly stole the show: He spoke every morning after breakfast. (He had asked to speak "early." However, instead of lecturing at the beginning of the Summer School, he ended up being the first speaker every day!) He talked about axiomatising the current concepts of programming. The interesting thing was that he not only combined this with correctness proofs of programs but also with elements of programming languages. It was further commendable to observe that he expressed many reservations about the limitations of his subject: he made it clear that, in all sorts of cases he did not regard it as realistic to supply a formal correctness proof. Sort of like "If you are challenged to supply a correctness proof, you should be able to do that, but it is not something you should do all the time."

Niklaus Wirth described Pascal and explained how his group succeeded in writing the first compiler in Pascal and making it compile itself:

> During the first week, N. Wirth from Zurich spoke for eight hours about his creation, the programming language Pascal. . .The Pascal implementation is a beautiful work that confirms my impression of him as one of Europe's most competent programmers.

> The Pascal compiler was first written in Pascal. It was then in a couple of weeks translated by hand into a low-level language. Two weeks after they first gained access to the [CDC 6600] computer, the Pascal compiler compiled itself.

Dijkstra gave a mixed review of my performance:

> The status of Pascal was not only enhanced by Hoare's experiment with axiomatization—Hoare has played an important role in the Pascal draft—but also because Per Brinch Hansen used it to describe operating system principles. In the past year, Brinch Hansen has...worked on a book about "Operating System Principles" which he summarized in his talks. The text was fine, but was not well presented. As a speaker he is a bit monotonous, and he did not feel at ease: it obviously bothered him that he had to borrow so much from me and appeared in his search for his own position somewhat argumentative.

That was not quite right. I admired Dijkstra and was delighted to acknowledge his fundamental ideas. I was simply an inexperienced speaker who felt nervous about lecturing in English to an audience that included some of the world's leading computer scientists. When I returned to Marktoberdorf two years later, my beginner's problems had disappeared.

Perlis was in fine form. During one of his lectures, he told a hilarious story about a guy, who wanted to be buried in his Cadillac. As the car with the dead man was lowered into the enormous grave, one mourner said to another: "Man, that's living!"

The contrast between the rigorous scientist, Edsger Dijkstra, and the easygoing Alan Perlis was striking:

> I found the great Alan J. Perlis very poor. He speaks with incredible ease if not to say bluff; his jokes are the most carefully prepared part of his presentation, and it is a pleasure to listen to him, as long as you don't listen too carefully to what he says. Unfortunately, part of his audience did just that...His actual subject was LCC, a conversational language developed at Carnegie-Mellon. But this was especially difficult to defend in this company, since it flies in the face of the ideas of program structuring preached by Wirth, Hoare and me...I have never seen

> him so insecure; perhaps it had to do with his departure from Carnegie-Mellon the previous week to join Yale University. . .and start a new life. Professionally he did not make a good impression.

In the interest of fairness, I can't resist quoting another epigram by Perlis:

> When a professor insists that computer science is X but not Y, have compassion for his graduate students.

Over the years, it became a tradition at the Summer Schools to visit Neuschwanstein Castle, built in the style of a Medieval castle by King Ludwig II of Bavaria (1845–86). (Cinderella's castle in Disneyland is a small replica of Neuschwannstein.) In 1864, Ludvig became king at the age of 18. This shy young man tended to neglect the affairs of state in the capital Munich, preferring instead to live in the Alps, where he spent a fortune building castles. Eventually, a group of politicians decided to have the king declared insane and replace him by his uncle. However, the doctors who declared Ludvig to be mentally ill never examined him. On Sunday, June 13, 1886, the king was found drowned in Starnberg Lake. Walking through his castle, I was looking at a massive, handcarved wooden horse sleigh, when it suddenly struck me: In 1882, this romantic dreamer was building a fairy tale castle, while Edison was demonstrating his electric light in Munich, 50 miles away!

⋆ ⋆ ⋆

The Marktoberdorf Summer School was followed by a Seminar on Operating Systems Techniques held at the Queen's University in Belfast, Northern Ireland, between August 30 and September 3. The speakers included some of England's leading operating system designers. David Hartley gave a reappraisal of the Atlas I system, which had pioneered multiprogramming ten years earlier. Sandy Fraser and Roger Needham spoke about the Atlas II system (known as Titan). Tony Hoare and his coworker Mike McKeag presented survey papers on various aspects of operating systems. And I outlined my operating system text again (Brinch Hansen 1971).

The threat of terrorist bombings had discouraged several contributors, including Dijkstra, from attending the seminar. At the time, bombings were frequent in downtown Belfast. In my hotel room, I heard several faint explosions at night. During a presentation at the seminar, a female student entered the small lecture room, put her briefcase on the floor and left the

room. I thought she might have left it there to go to the bathroom. But Tony immediately rushed over, looked inside the briefcase, picked it up and ran after her. I remember thinking: "He is awfully nosy." Then it dawned on me that he was worried she might be a terrorist trying to blow us up. On another occasion, we were driving in a tourbus outside the city when the radio warned listeners not to be worried about sonic booms that might occur that day during a testflight of the Concorde supersonic plane. The bus erupted in nervous laughter.

At an official reception in the City Hall, we were all waiting in line to shake hands with The Right Honourable Lord Mayor of Belfast. I was impressed by his technique: as soon as the mayor grabbed my hand, he pulled me forward to keep the line moving.

At the final banquet, I was seated next to Tony Hoare. After welcoming everybody he sat down, and I looked forward to a relaxing dinner conversation after a long day of discussions. However, as soon as he sat down, Tony started writing on a paper napkin, saying to me: "Let P be a process which satisfies the following condition..." I remember thinking: Let this be a lesson to anyone who doesn't realize that the leaders of their field never take a break—they think all the time, and, if you don't do the same, you will never be one of them.

In Marktoberdorf and Belfast, Tony and I discussed the future of programming languages. The trend was clear. Since 1960, high-level programming languages had replaced machine coding in one application after another: scientific computing (Fortran and Algol 60), business data processing (Cobol), and compiler design (Pascal). It was obvious to us that the next challenge was to invent programming notation for concurrent programs.

Now, in order to cooperate on common tasks, concurrent processes must be able to exchange data and timing signals through shared variables in memory. From the beginning, operating system designers recognized that multiprogramming will not work unless operations on shared data structures take place strictly one at a time. The reason is obvious: To understand what happens when, say, a process sorts an array, you have to assume that the sorting takes place without interference from other processes. If other processes can make arbitrary changes to the same array, while it is being sorted, it would indeed be a miracle if it ever ends up being sorted.

This is known as the *mutual exclusion* problem. In the RC 4000 multiprogramming system, we solved it by the well-known technique of disabling interrupts during the execution of monitor procedures. This guaranteed

that the computer would never interrupt an indivisible operation once it was started.

Dijkstra had shown how to implement these *critical sections* (as he called them) using semaphores. (Hoare would rename them *critical regions*.) However, if semaphores are omitted or used incorrectly, concurrent processes will interfere with one another in time-dependent ways that depend on the relative speeds of the processes. And those speeds can be influenced by completely unrelated factors, such as the speed at which terminal users respond to messages, or the occasional delays when printers run out of paper. Every time, you execute a time-dependent program, it will produce different results—even if the input is the same. It behaves like a universe in which Newton's laws have been replaced by random, unpredictable events. Since you cannot reproduce the output of such a program, it is often impossible to locate time-dependent programming errors by testing.

At the Belfast symposium, Hoare (1971) presented a first attempt to extend programming languages with abstract notation for process synchronization. He proposed that the error-prone semaphores should be replaced by language notation that would enable a compiler to recognize shared variables and check that they are used inside critical regions only.

Since it is awkward to talk about notation without showing it, I will yield to temptation and show you what he did. Take, for example, a mailbox that can transmit one integer at a time from one process to another. In a Pascal-like notation, the mailbox is declared as a shared data record with two components—a message slot of type integer and a boolean indicating whether the mailbox currently is empty or not:

```
var mailbox: shared record
               slot: integer; empty: boolean
             end;
```

A sending process waits until the mailbox is empty before putting a message into the mailbox:

```
with mailbox when empty do
begin slot := message; empty := false end
```

A receiving process waits until the mailbox is non-empty before removing a message from the mailbox:

```
with mailbox when not empty do
begin received := slot; empty := true end
```

Hoare's notation suppresses implementation details (such as the synchronization mechanism and scheduling policy used) and shows only the essence of the problem: sending and receiving are critical regions on the shared mailbox. Since these operations are delayed until the mailbox is in a suitable state (empty or full), I named them *conditional critical regions*.

Hoare's paper was as an eye-opener for me: It was my introduction to *the difficult art of language design*. The idea of preventing time-dependent errors by compile-time checking struck me as magical at a time when multiprogramming systems relied exclusively on run-time checking of variable access.

I had already received an earlier draft of Hoare's paper and had used conditional critical regions in my own book manuscript. However, at the Belfast symposium, I expressed some reservations from a software designer's point of view (Brinch Hansen 1971):

> 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.

In his presentation, Hoare responded to my criticism (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.

Hoare's proposal was based on the programming style used in Dijkstra's THE system, where critical regions were scattered throughout the program text. At Marktoberdorf, Dijkstra briefly outlined an alternative model, where a shared variable and its critical regions were combined into a separate process, known as a "secretary." This server process would own the shared variable and execute the critical regions on request from client processes.

Dijkstra had implemented his multiprogramming system as cooperating processes communicating through *shared variables* in *unprotected memory*. From his point of view, the idea of combining shared variables and critical regions into server processes 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 secretary concept, which Dijkstra sketched informally, had already been used since 1969 in the RC 4000 system under the name of a "conversational process" (Brinch Hansen 1969a). This was a server process that could be in the middle of conversations with several client processes at the same time. In the RC 4000 system, the basic operating system was a conversational process that spawned other processes in response to messages from operator consoles. If the basic 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 serve other clients. Since it was impossible to predict when the system would respond to a request, a conversational process was an early example of the use of *nondeterministic communication*.

Dijkstra may well have been influenced by my RC 4000 manual, which explained the purpose and use of conversational processes. Mike McKeag (1972–73) would demonstrate the similarity of these ideas by using the RC 4000 message primitives to outline simple secretaries for the well-known synchronization problems, known as the "bounded buffer," the "dining philosophers," and the "readers and writers."

Be that as it may. The important thing was that a *resource manager* was a well-known programming technique in the form of a *basic monitor*, invoked by supervisor calls, or a *conversational process* (a "secretary"), invoked by message passing. Our future efforts would be focussed on extending programming languages with an elegant notation for this *monitor concept* (as I called it).

⋆ ⋆ ⋆

I now returned to Carnegie-Mellon to finish my book. Today, when all operating system texts cover the same material, it may look like a well-defined task. But I had no models to imitate. It wasn't even obvious to me, what the sequence of chapters should be. To write the first systematic book on operating systems I would have to shape the field.

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.

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.

In 1971, when I had been working on my book for almost a year, the National Academy of Engineering summarized the state of affairs at the time:

> 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. The best source material is found in technical papers that frequently are hard to locate, understand, or correlate. (Cosine Report 1971)

Indeed! I still remember one instance of the slow, methodical task of trying to make sense of the available sources. From Regnecentralen's excellent library I brought a one-foot high stack of photocopies of papers that included descriptions of memory allocation techniques used in various systems. In our house in Pittsburgh, I read all of it and eliminated half of it as poorly written or of minor interest. Then I read the remaining half again and cut that in half, and so on, until I finally had reduced it to a dozen papers. At that point, my mind was so saturated with the subject that I began to see a reasonable pattern for my chapter on memory management.

By the time I outlined my book in Belfast, I had written drafts of half of the eight chapters. And, I had 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*.

⋆ ⋆ ⋆

In the spring of 1972 it was show time! I sent my finished manuscript to Karl Karlstrom, the famous computer science editor at Prentice-Hall, who asked Jim Horning, University of Toronto, what he thought of it. On May 1, Jim answered:

> Dear Karl:
>
> This is a preliminary report on Operating System Principles by Per Brinch Hansen, that I am reading with great interest. I think you should definitely try to get the book, and to give it the speedy publication which Brinch Hansen wants.
>
> This is a good book, perhaps even a great book, better than anything I have seen on operating systems. Its style and clarity makes it suitable for third-year undergraduate students, yet it is sufficiently comprehensive for use in a graduate course. I

> have not noted any technical errors, and remarkably few errors of presentation.
>
> I am eager to see the book come out soon, but not at the cost of cutting corners. This book will be around for a long time, and it should be handsomely done.
>
> Jim Horning

Eleven days later, I signed a publishing agreement. I also sent complete copies of the manuscript to Edsger Dijkstra and Tony Hoare.

In July 1973, Prentice-Hall published *Operating System Principles* (Brinch Hansen 1973a), and I was again invited to lecture at the Summer School in Marktoberdorf. This time, Dijkstra (1973) wrote:

> Personal reasons prevented M. Griffiths of Grenoble from speaking as scheduled, but we were lucky in having Per Brinch Hansen (California Institute of Technology) as a substitute. He went through the highlights of his recently published book "Operating System Principles" and he did that much, much better than two years ago, when he covered the same material in a very biased and even aggressive manner. Now he gave a neat, balanced survey. It is a pity that he has a very monotonous voice; it is really soporific and now I cannot even read one of his publications without hearing it.

Two years later, Peter Naur reviewed my book. Please, forgive me for indulging myself. It was, after all, Peter who had shown me that computer programming should be guided by sound principles and clear writing. In BIT 15 (1975), he wrote:

> It would be improper not to start this review with a warning: the reviewer is biased in favour of this book. In fact as mentioned in several remarks in the book, the work and attitude embodied in it have been influenced positively by the work with projects around 1964 for which your present reviewer was partially responsible. Your reviewer is unable to suppress a feeling of joy that the seeds sown at that time have borne this book as its fruit.
>
> It needs hardly to be said that the subject of this book, operating systems, are complex constructions. They are the outcome of the efforts of computer designers and programmers to apply

> their powerful tools to their own central problem: to control the use of computers in a flexible and efficient manner. True to its title, this book is a deliberate attempt to get to grips with this complexity through a concentration on principles. For this reason the book addresses the advanced reader, who is already familiar with at least some of the existing operating systems. In fact, the general character of the problem is not described in depth, but merely reviewed in the 22 pages of chapter 1.
>
> The presentation is generally at a very high level of clarity, and gives evidence of deep insight. In pursuing his principal 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. The main terms used in the text are explained in a vocabulary at the end of the book.
>
> As said explicitly in the introduction, the book presents one man's view of the subject. Thus, in particular, only one actual system, the RC 4000 multiprogramming system, is described in detail. Even so the author is very open to the ideas of others, and the text is scattered with references to work of others. The author must also be credited that he is able to maintain a critica1 attitude even to his own insight and solutions. Thus the discussion of the RC 4000 multiprogramming system ends with a detailed critical review of both advantages and disadvantages of the system. As a whole the book is not of the kind that is designed to provide the complete, closed basis for a course, but rather of the kind that tries to open the student's mind to the open ends, the unsolved problems, and the divergence of views, as expressed in other literature.
>
> In summary, this book treats the problems lying within its limitations with deep insight, a keen sense of underlying, general principles, and excellent clarity of exposition. It will be suitable for use in advanced computer science education and for the systems programmer and research worker. Highly recommended.
>
> Peter Naur

Believe it, or not, while writing my first book, it never occurred to me that I had any competition in this country of 250 million people. Once more,

youthful ignorance worked to my advantage. A year after my book came out, there were already several other texts on operating systems. Now, of course, there must be hundreds. Over the years, Karlstrom contracted with foreign publishers to translate my book into Japanese (1976), German (1977), Czech (1979), Polish (1979), and Serbo-Croatian (1982).