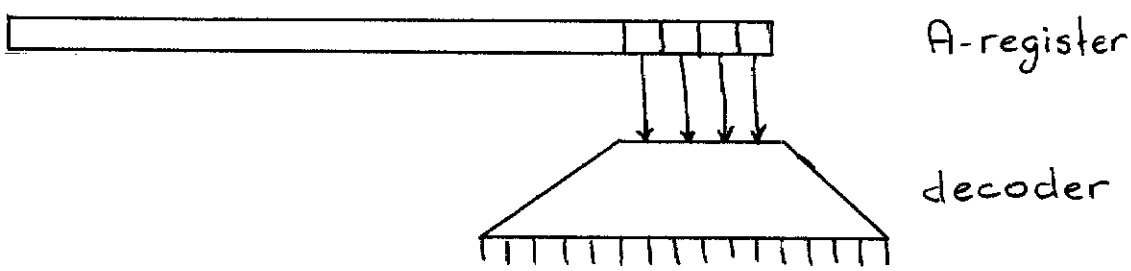
My recollections of operating system design

From, say, 1956 to 1966, I was involved in aspects of what would now be called the design of "operating systems". At the time, those years were exciting, in retrospect they are interesting. They were exciting because we were faced with all sorts of challenges of which we did not know whether they could be met at all: sometimes we succeeded, sometimes we failed. And it is now interesting to see how long it took for some key concepts to emerge and how separable (and eventually separated) problems remained for many years intertwined, solely because they had presented themselves together. (For many years, for instance, non-determinacy and concurrency were always closely linked: the one was never considered without the other.)

In what follows, I shall try to provide my recollections with all the background information needed to understand the story.

The source of the problems

In Amsterdam, at the Mathematical Centre, I started programming for binary machines for which an electric typewriter with electromagnets under the keys was the primary output device. The first machine could operate 16 keys of the typewriter: the 10 decimal digits, plus, minus, period, space, tabulate and NLCR (= New Line Carriage Return), the second one had access to the entire keyboard. The type instruction pulled the key identified by the 4 (6) least significant digits of the A-register (= main Accumulator); these bits were chosen because that was the place where the conversion process from binary to decimal representation would produce each time the binary representation of the next digit to be typed.

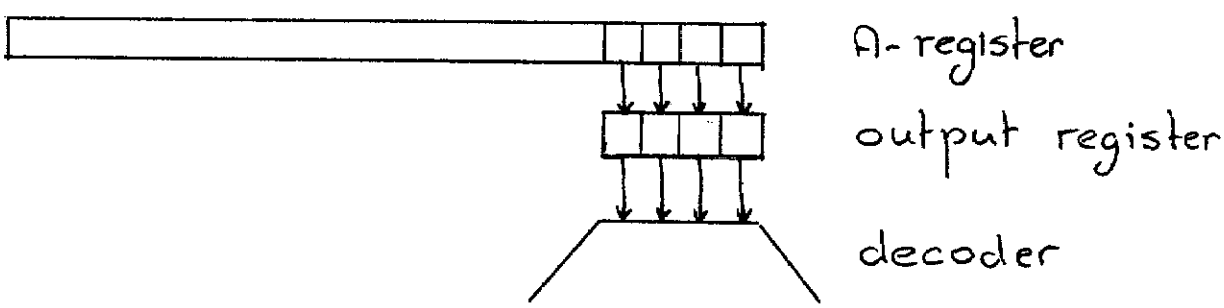


Between the 4 flip-flops of the A-register and the typewriter was a "decoder", i.e., a

device with 4 input wires and 16 output wires: the combination of the signals on the 4 input wires would determine which one of the 16 output wires could activate its electromagnet.

In the above arrangement, the A-register is either available for calculation or has to control the decoder, and thus the machine as a whole is a strictly sequential device in which calculating and typing alternate.

Mechanical devices as typewriters being as slow as they are, the decoder needs for each character typed its input for 100 ms, which by electronic standards is a long period of time: it is a pity to force for that whole period the calculator to idleness. This can be remedied for the price of introducing a 4-bit "output register":



At the beginning of the execution of the type instruction, the least-significant bits of A are copied into the output register at electronic

speed, and then, during the next 100 ms, two independent processes take place concurrently: having the A-register at its full disposal, the unhampered calculator continues to calculate at electronic speed, while under control of the output register the mechanical typewriter types the character. (The maximum gain in speed —rarely achieved in practice— is a factor of 2, but the price of an extra output register is not negligible when each flip-flop requires 2 radio valves. I don't remember whether my first machine had such an output register, but the second machine had one of 6 bits. Sheer luxury!)

In the above description, I skipped a minor problem: I said that during those 100 ms "having the A-register at its full disposal, the unhampered calculator continues to calculate" but what if, during those 100 ms it tries to execute another type instruction? This would change the contents of the output register and hence interfere with the process of printing the current character. Perhaps even typewriter arms could get entangled!

What to do? We can leave the calculator unhampered but burden the programmer with the obligation not to write - not even accidentally! - programs that might entangle typewriter arms, but this option is generally considered to be unacceptable. The arrangement would violate the reasonable requirement that no (cheap) program should be able to damage your (expensive) machine. It would have a further disadvantage of a type that was recognized only later: suppose that a technical improvement would speed up the calculator by a factor of 2, then, as a result, a formerly acceptable program could now damage the typewriter by executing a next type instruction too soon. (This is called "a real-time consideration"; we'll return to this issue later.)

The next alternative is a little hardware extension that conditionally hampers the calculator indeed: during the 100 ms, the calculator can work at full speed unless it tries to execute a type instruction, for in that case it is instantly frozen and only allowed to proceed after the 100 ms

have expired. (Depending on the mechanical properties of the typewriter, it could be that a period of 100 ms is not homogeneously appropriate: because of movement of the carriage, TAB and NLCR might need more time. If so, the period's dependence on the "character" typed should be incorporated in the hardware extension if we want to keep the programming of the calculator free of real-time considerations.)

I would like to stress that with this conditional freezing of the calculator, we have achieved CCC (= Completely Concealed Concurrency) in the sense that our two machines (i.e., with and without the output register) are now functionally equivalent in the sense that, fed with the same program, they will produce exactly the same output. As long as speed of execution is unobservable, it is impossible to determine which of the two machines did execute your program. (Hence the name CCC.)

Thus CCC improved the efficiency by allowing the simultaneous activity of temporarily independent system components

without any of the potential complications of concurrency, and that was its great virtue, but the comfortable invisibility of concurrency had a high price.

The first machine in which I saw that price paid in full was the Telefunken TR4. It incorporated CCC on a scale much more grandiose than the 4- or 6-bit output register I just described. The store of the TR4 was partitioned into 8 or 16 banks - I don't remember the exact number - and the central processor could initiate a communication action - say reading or punching a punched card - involving information between the communication device and a specified storage bank. During the information transport the CPU could continue calculating as long as it did not address that specified bank; when it did, it would be held up until the communication action had been completed. Several communication actions could be active simultaneously provided they each had their own bank. It was CCC in its full glory.

Consider now a program that has to

process many punched cards, while the order in which this happens is irrelevant — say we are adding voting counts —. Suppose furthermore that the machine is fast enough to engage all its card readers in this process; in that case we would like to do that, with each card reader running at its maximum speed most of the time. Note (i) that the card readers, being independent pieces of equipment, cannot be expected to have the same speeds, and (ii) that a card reader's speed need not be constant, as it is zero when the input tray is empty.

To maximize throughput, we would like each reader to read at its maximum speed most of the time, which means that the calculator should issue for each reader the next communication command as soon as possible after the completion of its preceding communication action. In particular, with the calculator idling for lack of input, one would like it to continue as soon as one of the readers has completed its input action, but with CCC, the calculator has no means of identifying that input stream: the invisi-

bility of concurrency has made the notion of "first-come-first-served" inapplicable. The moral of the story was that, essentially for the sake of efficiency, concurrency should become somewhat visible. It became so, and then, all hell broke loose.

Pandora's box

Enabling the central processor to react to the relative timing of the progress of the various communication devices created a host of new problems, so new that we didn't really know how to think and talk about them and what metaphors to use. To mention one example, people were used to view the control of the whole installation as localized in the central processor: wasn't that the component that issued its commands to the communication devices, wasn't that what the popular literature called "the electronic brain"? Now, suddenly it looked as if the central processor might have to serve a peripheral device at its bidding! Weren't we trying to switch from one master with many slaves to one slave having to serve

many masters (who might give incompatible orders)? People felt like facing a revolution they did not understand. Needless to say, the widespread use of anthropomorphic terminology only aggravated the situation. [Here I must confess to having prolonged the confusion somewhat by later introducing "directors" and "secretaries"; fortunately this was in a paper that has largely been ignored.]

In retrospect, the problems were in roughly three areas:

(i) The basic mechanics of switching the central processor from one task to another,

(ii) The strategy for scheduling the central processor and the scope of its commitment,

(iii) The logical problems that emerge when a number of resources are shared by a number of competitors,

and all that without theories or agreed criteria that could assist you in your choice between the options. Those years taught me that the ability to detect timely that some theory is needed is crucial for successful software design.

The simplest idea, which was often suggested for process control, was to have one "physical machine" simulate a small number of "logical machines", each coupled to its own device, such that the state of the device would determine whether the corresponding logical machine could proceed. Each logical machine was given its own priority and at each moment the physical machine would simulated of the logical machines that could proceed the one with the highest priority. The priorities and the way they controlled the allocation of the single processor were built-in hardware features of the physical machine.

The idea was that the logical machine of the highest priority would be coupled to the device with short, urgent, frequent messages, while those of lower priority would be coupled to devices that could be served more leisurely. If so desired, the one of the lowest priority could be devoted to an interesting "background" computation, say the generation of prime numbers, because it would be a pity to leave that expensive processor idling.

The above arrangement was usually all right as long as the logical machines were coupled to independent experiments, but as soon as logical machines had to communicate with each other, for instance because they dealt with different aspects of the same experiment, things became very difficult. In principle, logical machines could communicate with each other by writing and reading it the common store, but as a rule safe communication was very tricky if not impossible. It was quite common to take the risk that things would go wrong, as they would when the central processor was reallocated exactly at an inopportune moment, and in many an installation the risk had been taken unconsciously. (The frequency with which today your desktop system crashes in an irreproducible manner, suggests that such bugs are still quite common.)

Furthermore the above arrangement turned out to be too rigid for general-purpose computing: it is quite easy to think of a process in which actions of very different urgency alternate. (What about a child that throws a ball into the

air when it feels like it and then has to catch the ball.)

More flexibility was achieved by the introduction of "probe instructions" that enabled the central processor to test whether a concurrent activity it had started earlier had, in the mean time, been completed. Such probes are tricky: while the answer "Yes" is reliable, the answer "No" can become obsolete as soon as it has been given. (The designers of the IBM360 of 1964 had still fallen into this trap, with the result that users of "command chaining" had to accept the risk that a communication command got lost!)

But the inclusion of probe instructions created all sorts of dilemmas. Since the execution of the probe instruction takes time, probing with high frequency could noticeably slow down the compution, while probing with low frequency could increase the computer's reaction time too much. But even if we can agree on a frequency that may not be the end of our problems. Say that you would like to probe every 100 instructions, do you insert a

probe instruction into a loop body of 10 instructions? If you do, you might probe with unnecessarily high frequency, if you don't, you might probe with undesirably low frequency. Your subroutine library presents another problem; you might find yourself tempted to introduce each subroutine in two versions, one with probe instructions and the other without.

The third arrangement, known as "the interrupt", circumvents all these dilemmas. While the computer calculates at full speed, a piece of dedicated hardware monitors the outside world for completion signals from communication devices. When a completion is detected, the program under execution is interrupted after the current instruction and in such a fashion that it can be resumed at a later moment as if nothing had happened, thus instantaneously freeing the central processor for a suddenly more urgent task. After the interrupt the processor would execute a standard program establishing the source of the interruption and taking appropriate action.

It was a great invention, but also a

Box of Pandora. Because the exact moments of the interrupts were unpredictable and outside our control, the interrupt mechanism turned the computer into a nondeterministic machine with a nonreproducible behaviour, and could we control such a beast?

Below I shall address the major issues as I now remember encountering them; unavoidably this is very much a description "in retrospect": a lot of the terminology I use now did not exist at the time nor were different issues as clearly separated as they are now. (The now common expression "separation of concerns" hadn't been coined yet.)

I had designed the basic software for the ARRA, the FERTA and the ARMAC, and was going to do the same for the X1, the next machine my hardware friends Bram J. Loopstra and Carel S. Scholten were designing. At the time I was quite used to programming (in machine code) for machines that didn't exist yet: at the time the hardware got operational, I would have all the communication programs ready. The machines could execute programs one instruc-

tion at a time so that one could inspect intermediate states when something went wrong.

Halfway the functional design of the X1, I guess early 1957, Bram and Carel confronted me with the idea of the interrupt, and I remember that I panicked, being used to machines with reproducible behaviour. How was I going to identify a bug if I had introduced one? After I had delayed the decision to include the interrupt for 3 months, Bram and Carel flattered me out of my resistance, it was decided that an interrupt would be included and I began to study the problem. To start with I tried to convince myself that it was possible to save and restore enough of the machine state so that, after the servicing of the interrupt, under all circumstances the interrupted computation could be resumed correctly.

This turned out to be a very rewarding exercise, for not only was I unable to prove the possibility, I could demonstrate the impossibility as well. Bram and Carel changed their design accordingly, after which I designed a protocol for saving

and restoring that could be shown always to work properly. For all three of us it had been a very instructive experience. It was clearly an experience that others had missed because for years I would find flaws in designs of interrupt hardware.

I designed the interrupt handler and the asynchronous conversion routines — "the input/output programs" — for the basic peripherals each copy of the machine would be equipped with. It was the code that each copy of the machine would have wired in. It comprised slightly over 900 instructions, it was the work with which I earned my Ph.D. and no one would call it an operating system: the machine was envisaged as a uniprogramming system with concurrent activity of the peripherals, which can request the attention of the central processor.

An interlude

After the completion of these communication programs I was for a couple of

years otherwise engaged, first with writing other programs for the X1 - that was the name of the machine for which I designed the interrupt handler - and then thinking about compilers and ALGOL 60, which we implemented at the Mathematical Centre, essentially during the first half year of 1960.

As a consultant to Electrologica - the manufacturer of the X1 - I participated in discussions on how to attach punched-card equipment, and in the process I became very wary of what later would become known as "real-time problems". We called them "situations of urgency", i.e. the situations in which the computer had to react or to produce a result "in time". I remember, for instance, a device in which a punched card would pass, in order, a reading station, a punching station, and a final reading station. The role of the final reading station was simply to check that the card leaving the device had the proper holes, and that was no serious problem. The difficulty came from the combination of two circumstances, viz. (i) in most applications, the additional information to be punched into a card would depend on what had already been

punched into it, i.e. the computer would have to perform a problem-specific computation between the reading and the punching of the same card, and
(ii) once a card had been fed into the first reader, it could no longer be stopped and would inexorably pass the other stations so many milliseconds later.

We learned to distinguish between "essential urgency", where failure to be in time would derail the computation, and "moral urgency", where the failure would only hurt the efficiency (say in the form of a missed drum revolution). This was an important lesson for it taught us that in resource allocation - a problem we would tackle a few years later in its full glory - one could (and therefore should) separate the concerns of necessity and of desirability.

Connecting the machine to one card punch of the type I described was bad, connecting it to two of them was terrible. I found the combination of resource sharing and real-time constraints totally untractable and, slowly, the avoidance of urgency situations became a con-

scious goal. Choosing peripherals that could be stopped and introducing enough dedicated buffers often did the job.

But, as said, until mid 1960 I was otherwise engaged. In the summer of 1959, I was one of the many programmers who invented a stack to implement recursion. (This was EWD6, written during a holiday in "Het Familiehotel" in Paterswolde, in the northern part of the Netherlands. The manuscript no longer exists.) That fall, I embarked on parsing problems, and then we implemented ALGOL 60 on the X1. At the time the ALGOL implementation was by far the most ambitious undertaking I had ever embarked upon, and I knew that I would never get the thing working unless the structure of the compiler and of the object programs were as clear, simple, and systematic as I could get them. It was my first effort at implementation and the result was not clean at all by later standards when bootstrapping had become a well-mastered technique, but as far as I was concerned that did not matter. I had learned that a programmer should never make something more complicated

than he can control. (Of course, the fact that the X1 I was implementing ALGOL 60 for had only 4K words of storage, helped to keep things simple.)

Multiprogramming

For IBM machines, the purpose of "operating systems" -and hence their name- was to assist the operators in juggling tape reels and decks of cards, a purpose that never appealed to me since I have always been a paper-tape person.

Other cultures were driven by the perception that, at least for the next decade, processors would be too expensive to let them idle half the time and stores fast enough to match the processor's speed would be relatively small, so that a multi-level store was something we should have to live with. Examples reflecting this vision were the Burroughs B 5000 and its successors (USA), the Ferranti/Manchester ATLAS (UK) and the Electrologica/Eindhoven X8 (NL), machines that in contrast to IBM's products, were

loved by their user communities. Besides the decade of syntax, the 60s also became the decade of demand paging and replacement algorithms, but in the summer of 1960, all that was still future for me.

I resumed my ponderings about the synchronization between independently clocked entities, and to begin with I did so, true to my past experience, in the context of a central processor coupled to all sorts of peripheral gear, and I still remember my excitement about an in retrospect totally trivial discovery.

I recall the coupling between the computer and its output typewriter with the output register between the computer's A-register and the typewriter's decoder, where the computer would be temporarily frozen if it tried to execute the next type instruction before the execution of the previous type instruction had been completed. The exciting discovery was one of symmetry: just as the computer could be temporarily frozen because the typewriter wasn't ready yet for the next typeaction, so could the typewriter

be temporarily frozen because the computer was not ready yet for the next type action; just as the computer could have to wait until the output buffer was empty, the typewriter could have to wait until the output buffer was filled.

To see precious milliseconds being lost because the expensive computer was being delayed by a cheap, slow peripheral, was most regrettable, but when that same cheap, slow typewriter was idle because there was nothing to be typed, we could not care less. The excitement came from the recognition that the difference in emotional appreciation was irrelevant and had better be forgotten, and that it was much more significant that from a logical point of view they were two instances of the same phenomenon. In retrospect, this insight should perhaps be considered more radical than it seemed at the time, for decades later I still met system designers who felt that maximization of the duty cycle of the most expensive system components was an important target.

With the symmetry between CPU and

peripheral clearly understood, symmetric synchronizing primitives could not fail to be conceived and I introduced the operations P and V on semaphores. Initially my semaphores were binary, reflecting that the output register I described earlier could be empty or filled. When I showed this to Bram and Carel, Carel immediately suggested to generalize the binary semaphore to a natural semaphore, a generalization that would greatly enhance the usefulness of the semaphore concept: it enabled us to generalize the one-character output register to an output buffer of any size. How valuable this generalization was we would soon discover.

The component taking care of the information transport between store and peripheral was called "a channel"; our channels could be active concurrently with the CPU and as a rule their activity was synchronized with the peripheral they served.

The simplest channels would accept one "command" at a time; this would be, say, the reading of a card, the writing

of a block on a magnetic tape or the reading of half a track of information from the drum. For an act of communication the CPU would specify its parameters at a dedicated location and then would send a start signal to the channel. After the act of communication, the raising of an interrupt flag would signal the completion to the CPU, which was then free to present a next communication command to the channel.

Many peripherals could be designed in such a way that they did not confront the CPU with real-time obligations, with situations of "essential urgency", for instance, the magnetic tape unit would stop automatically if the next command to write a block of information would not come soon enough, other devices like card readers or drums would just miss a cycle or a revolution. So far, so good, but the problem was, that these same devices would often now confront the CPU with situations of "moral urgency": it was such a pity that it really hurt if the channel did not get its next command in time.

The solution was to allow the CPU

to build up a queue of commands and to enable the channel to switch without CPU intervention to the next command in the queue. Synchronization was controlled by two counting semaphores in what we now know as the producer/consumer arrangement: the one semaphore, indicating the length of the queue, was incremented (in a V) by the CPU and decremented (in a P) by the channel, the other one, counting the number of unacknowledged completions, was incremented by the channel and decremented by the CPU. [The second semaphore being positive would raise the corresponding interrupt flag.]

In the IBM/360, which appeared a year later, such queuing was done by "command chaining": in the chain of queued commands, the last one carried an end marker, and attaching a new command was done by overwriting that end marker by a pointer to the command to be attached. But the design contained two serious flaws.

If at the moment of attachment the channel was still active, the new command would be executed in due time, otherwise

the channel had to be reactivated. To establish whether this obligation was present, the instruction code contained test instructions testing whether channels were active or not, but the trouble with such a test instruction is that when a channel is reported to be still active, that information can be obsolete a microsecond later. In the IBM/360 the result was that, when a new command had been attached while before the attachment the channel was observed to be active and after the attachment it was passive, this information was insufficient to determine whether the channel had deactivated itself before or after the attachment, i.e. whether the last command had been executed or not.

Reporting completions was similarly defective. Command chaining had introduced channels that in due time would report a sequence of completions, but instead of a counter there was only the binary interrupt flipflop to record a next completion. If an interrupt signal came before the previous one for that channel had been honoured, it would just get lost! They had traded the minor evil of moral urgency for the major one of essential urgency.

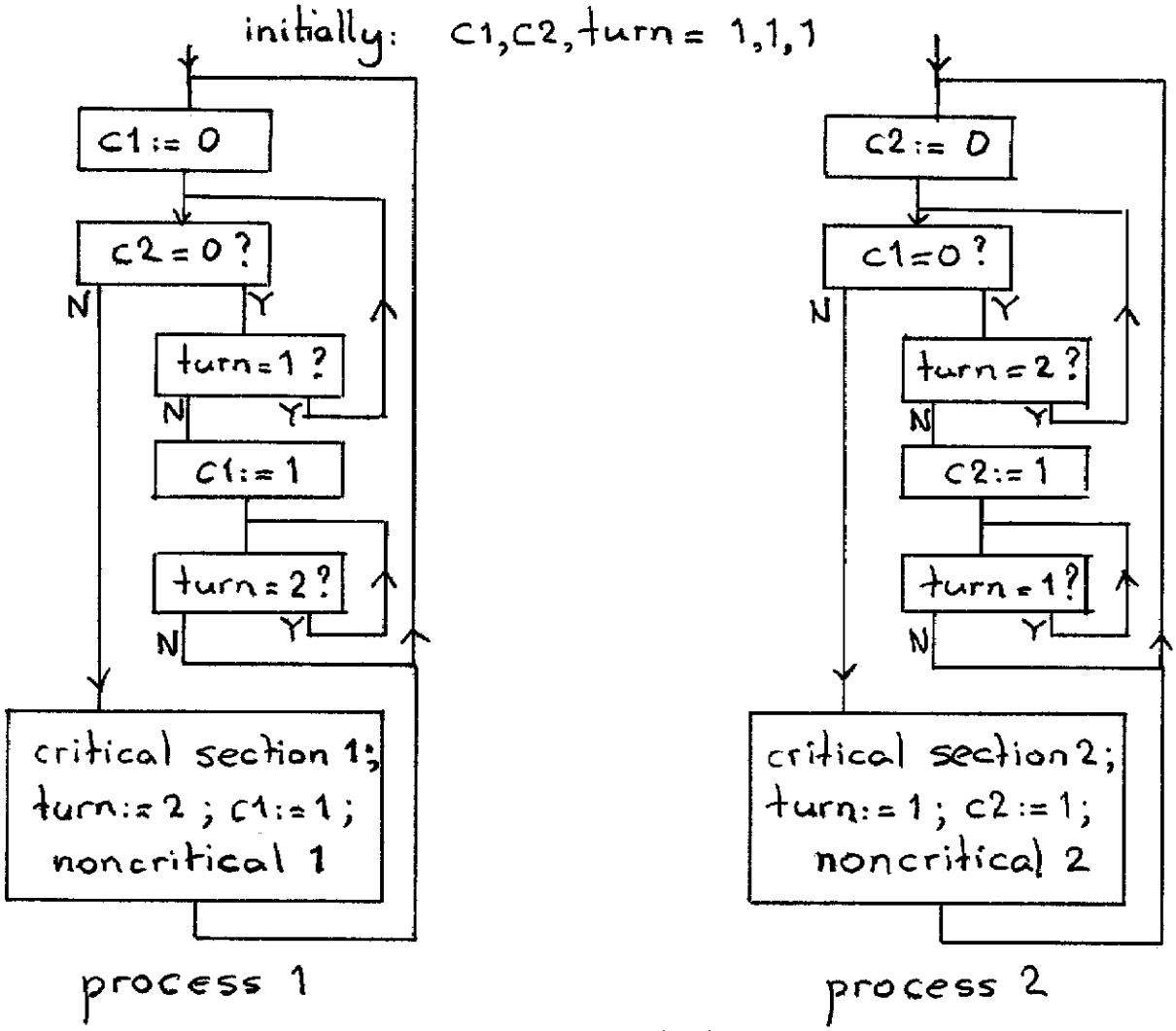
At the time I was shocked by the fact that the major product of the world's largest and most powerful computer manufacturer could contain such a serious blunder. Later I realized that to a certain extent I myself was to be blamed, for when I invented the semaphores and the synchronizing primitives, I did not publish my invention. We did not keep it secret, we lectured freely about the topic, but did not publish it until the end of the decade. This was a self-imposed censorship. I knew that it would determine the channel organization of the EL X8, the next machine of the NV Electrologica, of which I was a consultant. Trusting that the lovely channel organization of the EL X8 would be a major selling point of the machine, I did not broadcast my invention among the competition. At that time I clearly still believed in a positive correlation between a product's technical quality and its commercial success. In retrospect I am sorry that I postponed widespread publication until 1967 and think that I would have served mankind better if I had enabled The Evil One to improve its product.

For economic reasons one wants to avoid in a multiprogramming system the so-called "busy form of waiting", in which the central processor is unproductively engaged in the waiting cycle of a temporarily stopped program, for it is better to grant the processor to a program that may actually proceed. This economic consideration was in 1961/62 a strong incentive for the introduction of the P/V-synchronization mechanism, which made it explicit to the scheduler when a program was no candidate for processor time. An in retrospect more profound incentive, however, came from an experience I had had in 1959 at the Computation Department of the Mathematical Centre in Amsterdam.

I had invented what, in that environment at least, was a new type of problem. Consider two cyclic processes, each consisting of an alternation of "a critical section" and "a noncritical section". Was it possible to synchronize these two processes such that at any moment at most 1 of the processes would be engaged in its critical section? The way in which the processes could communicate was by atomic

reads and writes in a common store. After having convinced myself that —at least according to the State of the Art of those days— the answer was not obvious, I broadcast the problem to the members of the Computation Department for their general amusement and distraction. It was given that each execution of a critical section would only take finite amount of time and it was required that a process ready to execute its critical section would in due time get the opportunity to do so.

Many purported solutions were handed in, but on closer inspection they were all defective, and my friends realized that the problem was harder than they had suspected and that their simple "solutions" wouldn't do. Fortunately my friends didn't give up and they handed in "improved" versions of their earlier efforts. But as their designs became more and more complicated, it became harder and harder to construct a counterexample. I couldn't keep up with them and had to change the rules of this programming contest: besides a program I required a proof that that program had all the desired properties.



Th. J. Dekker's Solution

And then something profound and lovely happened. By analysing by what structure of argument the proof obligations could be met, the numerical mathematician Th. J. Dekker designed within a few hours the above solution together with its correctness argument, and this settled the contest. In the above solution, the pair "c1, c2" implements the mutual exclusion, while

"turn" has been introduced to resolve the tie when the two processes simultaneously try to enter their critical sections.

To capture the spirit of the time, I have presented Dekker's solution as a flow chart of the type we then used to design and document our programs. It looks very simple; there are only 3 commonly accessible bits, two, "c1 & c2", written by one process and inspected by the other, and a third one, "turn", which is written and read by both. But how great an invention it was is shown by the fact that the generalized problem for N processes, instead of for two, was only solved in 1965; the latter solution was also considered a tour de force.

The profound significance of Dekker's solution of 1959, however, was that it showed the rôle that mathematical proof could play in the process of program design. Now, more than 40 years later, this rôle is still hardly recognized in the world of discrete designs. If mathematics is allowed to play a rôle at all, it is in the form of a posteriori verification, i.e. by showing by usually mechanized mathe-

matics that the design meets its specifications; the strong guidance that correctness concerns can provide to the design process is rarely exploited. On the whole it is still "Code first, debug later" instead of "Think first, code later", which is a pity, but Computing Science cannot complain: we are free to speak, we don't have a right to be heard. And in later years Computing Science has spoken: in connection with the calculational derivation of programs -and then of mathematical proofs in general- the names of R.W. Floyd, C.A.R. Hoare, D. Gries, C.C. Morgan, A.J.M. van Gasteren and W.H.J. Feijen are the firsts to come to my mind.

At the time, the most tangible and most influential abstraction was the notion of loosely coupled sequential processes, their most essential property being their undefined speeds. You could not even assume individual speeds to be constant. This notion eliminated at one fell swoop all analogue arguments about time, all synchronization needed for logical reasons had to be coded explicitly, and consequently only discrete reasoning was needed to establish the system's correctness. Compared with all alternatives, this was a tremendous simplifica-

tion. This is now so obvious that it is hard to recognize today that at the time an invention was involved, but there was. It was before the term "layers of abstraction" had been coined, and I remember being blamed for ignoring speed ratios also when they were perfectly known: I was accused of throwing away possibly valuable information and had to defend my decision by pointing out that I was heading for a superior product in the sense that no reprogramming would be required when, say, the line printer would be replaced by a faster model.

For me the main virtue was that the logical correctness now no longer depended on real-time considerations: the term "tremendous simplification", which I used above, was no exaggeration. It made it possible for us to know so well what we were doing that there was no need "to debug our design into correctness". I have two pregnant memories of how differently others tried to work.

When the design of the THE Multiprogramming System neared its comple-

tion, the University's EL X8 was getting installed, but it had not been paid yet, and we hardly had access to it because the manufacturer had put it to the disposal of a crew of an American software house that was supposed to write a COBOL implementation for the EL X8. They were program testing all the time, and we let it be known that if occasionally we could have the virgin machine for a few minutes, we would appreciate it. They were nice guys and a few times per week we would get an opportunity for our next test run. We would enter the machine room with a small roll of punched paper tape, and a few minutes later we would leave the machine room with the output we wanted. I remember it vividly because when they realized what we were achieving, our minimal usage of the machine became more and more frustrating for them. I don't think their COBOL implementation was ever completed.

The other vivid memory is of what I heard about IBM's mechanical debugging aid. In order to make errors reproducible

they equipped the IBM/360 with a special hardware monitor; in the recording mode it would capture precisely where in the computation each interrupt took place, in the controlling mode it would force the computer to replay exactly the recorded computation, thus making intermediate results available for inspection. It struck me as a crude and silly way of proceeding, and I remember being grateful that lack of money had protected me from making that methodological mistake. I felt grateful and superior, and when a few years later it became clear that IBM couldn't get its OS/360 right, I was not amazed.

* * *

The period I more or less covered started while I was a staff member at the Computation Department of the Mathematical Centre in Amsterdam, it ended while I was professor of mathematics at the Eindhoven University of Technology. The graduates of our department got the official academic title of "Mathematical Engineer".

In the beginning I knew that programming posed a much greater intellectual

challenge than generally understood or acknowledged, but my vision of the more detailed structure of that vision was still vague. Halfway the multiprogramming project in Eindhoven I realized that we would have been in grave difficulties, had we not seen in time the possibility of definitely unintended deadlocks. From that experience we concluded that a successful systems designer should recognize as early as possible situations in which some theory was needed. In our case we needed enough theory about deadlocks and their prevention to develop what became known as "the banker's algorithm", without which the multiprogramming system would only have been possible in a very crude form. By the end of the period I knew that the design of sophisticated digital systems was the perfect field of activity for the Mathematical Engineer.

Austin, October 2000/April 2001

prof. dr Edsger W. Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
USA