The Structure of the "THE"-Multiprogramming System

Edsger W. Dijkstra
Technological University, Eindhoven, The Netherlands

A multiprogramming system is described in which all activities are divided over a number of sequential processes. These sequential processes are placed at various hierarchical levels, in each of which one or more independent abstractions have been implemented. The hierarchical structure proved to be vital for the verification of the logical soundness of the design and the correctness of its implementation.

KEY WORDS AND PHRASES: operating system, multiprogramming system, system hierarchy, system structure, real-time debugging, program verification, synchronizing primitives, cooperating sequential processes, system levels, input-output buffering, multiprogramming, processor sharing, multiprocessing
CR CATEGORIES: 4.30, 4.32

Introduction

In response to a call explicitly asking for papers "on timely research and development efforts," I present a progress report on the multiprogramming effort at the Department of Mathematics at the Technological University in Eindhoven.

Having very limited resources (viz. a group of six people of, on the average, half-time availability) and wishing to contribute to the art of system design—including all the stages of conception, construction, and verification, we were faced with the problem of how to get the necessary experience. To solve this problem we adopted the following three guiding principles:

(1) Select a project as advanced as you can conceive, as ambitious as you can justify, in the hope that routine work can be kept to a minimum; hold out against all pressure to incorporate such system expansions that would only result into a purely quantitative increase of the total amount of work to be done.

(2) Select a machine with sound basic characteristics (e.g. an interrupt system to fall in love with is certainly an inspiring feature); from then on try to keep the specific properties of the configuration for which you are preparing the system out of your considerations as long as possible.

(3) Be aware of the fact that experience does by no means automatically lead to wisdom and understanding; in other words, make a conscious effort to learn as much as possible from your previous experiences.

Presented at an ACM Symposium on Operating System Principles, Gatlinburg, Tennessee, October 1–4, 1967.

Accordingly, I shall try to go beyond just reporting what we have done and how, and I shall try to formulate as well what we have learned.

I should like to end the introduction with two short remarks on working conditions, which I make for the sake of completeness. I shall not stress these points any further.

One remark is that production speed is severely slowed down if one works with half-time people who have other obligations as well. This is at least a factor of four; probably it is worse. The people themselves lose time and energy in switching over; the group as a whole loses decision speed as discussions, when needed, have often to be postponed until all people concerned are available.

The other remark is that the members of the group (mostly mathematicians) have previously enjoyed as good students a university training of five to eight years and are of Master's or Ph.D. level. I mention this explicitly because at least in my country the intellectual level needed for system design is in general grossly underestimated. I am convinced more than ever that this type of work is very difficult, and that every effort to do it with other than the best people is doomed to either failure or moderate success at enormous expense.

The Tool and the Goal

The system has been designed for a Dutch machine, the EL X8 (N.V. Electrologica, Rijswijk (ZH)). Characteristics of our configuration are:

(1) core memory cycle time 2.5μsec, 27 bits; at present 32K;
(2) drum of 512K words, 1024 words per track, rev. time 40msec;
(3) an indirect addressing mechanism very well suited for stack implementation;
(4) a sound system for commanding peripherals and controlling of interrupts;
(5) a potentially great number of low capacity channels; ten of them are used (3 paper tape readers at 1000char/sec; 3 paper tape punches at 150char/sec; 2 teleprinters; a plotter; a line printer);
(6) absence of a number of not unusual, awkward features.

The primary goal of the system is to process smoothly a continuous flow of user programs as a service to the University. A multiprogramming system has been chosen with the following objectives in mind: (1) a reduction of turn-around time for programs of short duration, (2) economic use of peripheral devices, (3) automatic control

of backing store to be combined with economic use of the central processor, and (4) the economic feasibility to use the machine for those applications for which only the flexibility of a general purpose computer is needed, but (as a rule) not the capacity nor the processing power.

The system is not intended as a multiaccess system. There is no common data base via which independent users can communicate with each other: they only share the configuration and a procedure library (that includes a translator for ALGOL 60 extended with complex numbers). The system does not cater for user programs written in machine language.

Compared with larger efforts one can state that quantitatively speaking the goals have been set as modest as the equipment and our other resources. Qualitatively speaking, I am afraid, we became more and more immodest as the work progressed.

A Progress Report

We have made some minor mistakes of the usual type (such as paying too much attention to eliminating what was not the real bottleneck) and two major ones.

Our first major mistake was that for too long a time we confined our attention to "a perfect installation"; by the time we considered how to make the best of it, one of the peripherals broke down, we were faced with nasty problems. Taking care of the "pathology" took more energy than we had expected, and some of our troubles were a direct consequence of our earlier ingenuity, i.e. the complexity of the situation into which the system could have maneuvered itself. Had we paid attention to the pathology at an earlier stage of the design, our management rules would certainly have been less refined.

The second major mistake has been that we conceived and programmed the major part of the system without giving more than scanty thought to the problem of debugging it. I must decline all credit for the fact that this mistake had no serious consequences—on the contrary! one might argue as an afterthought.

As captain of the crew I had had extensive experience (dating back to 1958) in making basic software dealing with real-time interrupts, and I knew by bitter experience that as a result of the irreproducibility of the interrupt moments a program error could present itself misleadingly like an occasional machine malfunctioning. As a result I was terribly afraid. Having fears regarding the possibility of debugging, we decided to be as careful as possible and, prevention being better than cure, to try to prevent nasty bugs from entering the construction.

This decision, inspired by fear, is at the bottom of what I regard as the group's main contribution to the art of system design. 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 (occurring with a density of one error per 500 instructions), each of them located within 10 minutes (classical) inspection by the machine and each of them correspondingly easy to remedy. At the time this was written the testing had not yet been completed, but the resulting system is guaranteed to be flawless. When the system is delivered we shall not live in the perpetual fear that a system derailment may still occur in an unlikely situation, such as might result from an unhappy "coincidence" of two or more critical occurrences, for we shall have proved the correctness of the system with a rigor and explicitness that is unusual for the great majority of mathematical proofs.

A Survey of the System Structure

Storage Allocation. In the classical von Neumann machine, information is identified by the address of the memory location containing the information. When we started to think about the automatic control of secondary storage we were familiar with a system (viz. GIER ALGOL) in which all information was identified by its drum address (as in the classical von Neumann machine) and in which the function of the core memory was nothing more than to make the information "page-wise" accessible.

We have followed another approach and, as it turned out, to great advantage. In our terminology we made a strict distinction between memory units (we called them "pages" and had "core pages" and "drum pages") and corresponding information units (for lack of a better word we called them "segments"), a segment just fitting in a page. For segments we created a completely independent identification mechanism in which the number of possible segment identifiers is much larger than the total number of pages in primary and secondary store. The segment identifier gives fast access to a so-called "segment variable" in core whose value denotes whether the segment is still empty or not, and if not empty, in which page (or pages) it can be found.

As a consequence of this approach, if a segment of information, residing in a core page, has to be dumped onto the drum in order to make the core page available for other use, there is no need to return the segment to the same drum page from which it originally came. In fact, this freedom is exploited: among the free drum pages the one with minimum latency time is selected.

A next consequence is the total absence of a drum allocation problem: there is not the slightest reason why, say, a program should occupy consecutive drum pages. In a multiprogramming environment this is very convenient.

Processor Allocation. We have given full recognition to the fact that in a single sequential process (such as can be performed by a sequential automaton) only the time succession of the various states has a logical meaning, but not the actual speed with which the sequential process is

performed. Therefore we have arranged the whole system as a society of sequential processes, progressing with undefined speed ratios. To each user program accepted by the system corresponds a sequential process, to each input peripheral corresponds a sequential process (buffering input streams in synchronism with the execution of the input commands), to each output peripheral corresponds a sequential process (unbuffering output streams in synchronism with the execution of the output commands); furthermore, we have the "segment controller" associated with the drum and the "message interpreter" associated with the console keyboard.

This enabled us to design the whole system in terms of these abstract "sequential processes." Their harmonious cooperation is regulated by means of explicit mutual synchronization statements. On the one hand, this explicit mutual synchronization is necessary, as we do not make any assumption about speed ratios; on the other hand, this mutual synchronization is possible because "delaying the progress of a process temporarily" can never be harmful to the interior logic of the process delayed. The fundamental consequence of this approach—viz. the explicit mutual synchronization—is that the harmonious cooperation of a set of such sequential processes can be established by discrete reasoning; as a further consequence the whole harmonious society of cooperating sequential processes is independent of the actual number of processors available to carry out these processes, provided the processors available can switch from process to process.

System Hierarchy. The total system admits a strict hierarchical structure.

At level 0 we find the responsibility for processor allocation to one of the processes whose dynamic progress is logically permissible (i.e. in view of the explicit mutual synchronization). At this level the interrupt of the real-time clock is processed and introduced to prevent any process to monopolize processing power. At this level a priority rule is incorporated to achieve quick response of the system where this is needed. Our first abstraction has been achieved; above level 0 the number of processors actually shared is no longer relevant. At higher levels we find the activity of the different sequential processes, the actual processor that had lost its identity having disappeared from the picture.

At level 1 we have the so-called "segment controller," a sequential process synchronized with respect to the drum interrupt and the sequential processes on higher levels. At level 1 we find the responsibility to cater to the bookkeeping resulting from the automatic backing store. At this level our next abstraction has been achieved; at all higher levels identification of information takes place in terms of segments, the actual storage pages that had lost their identity having disappeared from the picture.

At level 2 we find the "message interpreter" taking care of the allocation of the console keyboard via which conversations between the operator and any of the higher level processes can be carried out. The message interpreter works in close synchronism with the operator. When the operator presses a key, a character is sent to the machine together with an interrupt signal to announce the next keyboard character, whereas the actual printing is done through an output command generated by the machine under control of the message interpreter. (As far as the hardware is concerned the console teleprinter is regarded as two independent peripherals: an input keyboard and an output printer.) If one of the processes opens a conversation, it identifies itself in the opening sentence of the conversation for the benefit of the operator. If, however, the operator opens a conversation, he must identify the process he is addressing, in the opening sentence of the conversation, i.e. this opening sentence must be interpreted before it is known to which of the processes the conversation is addressed! Here lies the logical reason for the introduction of a separate sequential process for the console teleprinter, a reason that is reflected in its name, "message interpreter."

Above level 2 it is as if each process had its private conversational console. The fact that they share the same physical console is translated into a resource restriction of the form "only one conversation at a time," a restriction that is satisfied via mutual synchronization. At this level the next abstraction has been implemented; at higher levels the actual console teleprinter loses its identity. (If the message interpreter had not been on a higher level than the segment controller, then the only way to implement it would have been to make a permanent reservation in core for it; as the conversational vocabulary might become large (as soon as our operators wish to be addressed in fancy messages), this would result in too heavy a permanent demand upon core storage. Therefore, the vocabulary in which the messages are expressed is stored on segments, i.e. as information units that can reside on the drum as well. For this reason the message interpreter is one level higher than the segment controller.)

At level 3 we find the sequential processes associated with buffering of input streams and unbuffering of output streams. At this level the next abstraction is effected, viz. the abstraction of the actual peripherals used that are allocated at this level to the "logical communication units" in terms of which are worked in the still higher levels. The sequential processes associated with the peripherals are of a level above the message interpreter, because they must be able to converse with the operator (e.g. in the case of detected malfunctioning). The limited number of peripherals again acts as a resource restriction for the processes at higher levels to be satisfied by mutual synchronization between them.

At level 4 we find the independent-user programs and at level 5 the operator (not implemented by us).

The system structure has been described at length in order to make the next section intelligible.

Design Experience

The conception stage took a long time. During that period of time the concepts have been born in terms of which we sketched the system in the previous section. Furthermore, we learned the art of reasoning by which we could deduce from our requirements the way in which the processes should influence each other by their mutual synchronization so that these requirements would be met. (The requirements being that no information can be used before it has been produced, that no peripheral can be set to two tasks simultaneously, etc.). Finally we learned the art of reasoning by which we could prove that the society composed of processes thus mutually synchronized by each other would indeed in its time behavior satisfy all requirements.

The construction stage has been rather traditional, perhaps even old-fashioned, that is, plain machine code. Reprogramming on account of a change of specifications has been rare, a circumstance that must have contributed greatly to the feasibility of the "steam method." That the first two stages took more time than planned was somewhat compensated by a delay in the delivery of the machine.

In the verification stage we had the machine, during short shots, completely at our disposal; these were shots during which we worked with a virgin machine without any software aids for debugging. Starting at level 0 the system was tested, each time adding (a portion of) the next level only after the previous level had been thoroughly tested. Each test shot itself contained, on top of the (partial) system to be tested, a number of testing processes with a double function. First, they had to force the system into all different relevant states; second, they had to verify that the system continued to react according to specification.

I shall not deny that the construction of these testing programs has been a major intellectual effort: to convince oneself that one has not overlooked "a relevant state" and to convince oneself that the testing programs generate them all is no simple matter. The encouraging thing is that (as far as we know!) it could be done.

This fact was one of the happy consequences of the hierarchical structure.

Testing level 0 (the real-time clock and processor allocation) implied a number of testing sequential processes on top of it, inspecting together that under all circumstances processor time was divided among them according to the rules. This being established, sequential processes as such were implemented.

Testing the segment controller at level 1 meant that all "relevant states" could be formulated in terms of sequential processes making (in various combinations) demands on core pages, situations that could be provoked by explicit synchronization among the testing programs. At this stage the existence of the real-time clock—although interrupting all the time—was so immaterial that one of the testers indeed forgot its existence!

By that time we had implemented the correct reaction upon the (mutually unsynchronized) interrupts from the real-time clock and the drum. If we had not introduced the separate levels 0 and 1, and if we had not created a terminology (viz. that of the rather abstract sequential processes) in which the existence of the clock interrupt could be discarded, but had instead tried in a nonhierarchical construction, to make the central processor react directly upon any weird time succession of these two interrupts, the number of "relevant states" would have exploded to such a height that exhaustive testing would have been an illusion. (Apart from that it is doubtful whether we would have had the means to generate them all, drum and clock speed being outside our control.)

For the sake of completeness I must mention a further happy consequence. As stated before, above level 1, core and drum pages have lost their identity, and buffering of input and output streams (at level 3) therefore occurs in terms of segments. While testing at level 2 or 3 the drum channel hardware broke down for some time, but testing proceeded by restricting the number of segments to the number that could be held in core. If building up the line printer output streams had been implemented as "dumping onto the drum" and the actual printing as "printing from the drum," this advantage would have been denied to us.

Conclusion

As far as program verification is concerned I present nothing essentially new. In testing a general purpose object (be it a piece of hardware, a program, a machine, or a system), one cannot subject it to all possible cases: for a computer this would imply that one feeds it with all possible programs! Therefore one must test it with a set of relevant test cases. What is, or is not, relevant cannot be decided as long as one regards the mechanism as a black box; in other words, the decision has to be based upon the internal structure of the mechanism to be tested. It seems to be the designer's responsibility to construct his mechanism in such a way—i.e. so effectively structured—that at each stage of the testing procedure the number of relevant test cases will be so small that he can try them all and that what is being tested will be so perspicuous that he will not have overlooked any situation. I have presented a survey of our system because I think it a nice example of the form that such a structure might take.

In my experience, I am sorry to say, industrial software makers tend to react to the system with mixed feelings. On the one hand, they are inclined to think that we have done a kind of model job; on the other hand, they express doubts whether the techniques used are applicable outside the sheltered atmosphere of a University and express the opinion that we were successful only because of the modest scope of the whole project. It is not my intention to underestimate the organizing ability needed to handle a much bigger job, with a lot more people, but I should like to ven-

ture the opinion that the larger the project, the more essential the structuring! A hierarchy of five logical levels might then very well turn out to be of modest depth, especially when one designs the system more consciously than we have done, with the aim that the software can be smoothly adapted to (perhaps drastic) configuration expansions.

Acknowledgments. I express my indebtedness to my five collaborators, C. Bron, A. N. Habermann, F. J. A. Hendriks, C. Ligtmans, and P. A. Voorhoeve. They have contributed to all stages of the design, and together we learned the art of reasoning needed. The construction and verification was entirely their effort; if my dreams have come true, it is due to their faith, their talents, and their persistent loyalty to the whole project.

Finally I should like to thank the members of the program committee, who asked for more information on the synchronizing primitives and some justification of my claim to be able to prove logical soundness a priori. In answer to this request an appendix has been added, which I hope will give the desired information and justification.

APPENDIX

Synchronizing Primitives

Explicit mutual synchronization of parallel sequential processes is implemented via so-called "semaphores." They are special purpose integer variables allocated in the universe in which the processes are embedded; they are initialized (with the value 0 or 1) before the parallel processes themselves are started. After this initialization the parallel processes will access the semaphores only via two very specific operations, the so-called synchronizing primitives. For historical reasons they are called the P-operation and the V-operation.

A process, "Q" say, that performs the operation "P (sem)" decreases the value of the semaphore called "sem" by 1. If the resulting value of the semaphore concerned is nonnegative, process Q can continue with the execution of its next statement; if, however, the resulting value is negative, process Q is stopped and booked on a waiting list associated with the semaphore concerned. Until further notice (i.e. a V-operation on this very same semaphore), dynamic progress of process Q is not logically permissible and no processor will be allocated to it (see above "System Hierarchy," at level 0).

A process, "R" say, that performs the operation "V (sem)" increases the value of the semaphore called "sem" by 1. If the resulting value of the semaphore concerned is positive, the V-operation in question has no further effect; if, however, the resulting value of the semaphore concerned is nonpositive, one of the processes booked on its waiting list is removed from this waiting list, i.e. its dynamic progress is again logically permissible and in due time a processor will be allocated to it (again, see above "System Hierarchy," at level 0).

COROLLARY 1. *If a semaphore value is nonpositive its absolute value equals the number of processes booked on its waiting list.*

COROLLARY 2. *The P-operation represents the potential delay, the complementary V-operation represents the removal of a barrier.*

Note 1. P- and V-operations are "indivisible actions"; i.e. if they occur "simultaneously" in parallel processes they are noninterfering in the sense that they can be regarded as being performed one after the other.

Note 2. If the semaphore value resulting from a V-operation is negative, its waiting list originally contained more than one process. It is undefined—i.e. logically immaterial—which of the waiting processes is then removed from the waiting list.

Note 3. A consequence of the mechanisms described above is that a process whose dynamic progress is permissible can only loose this status by actually progressing, i.e. by performance of a P-operation on a semaphore with a value that is initially nonpositive.

During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores.

Mutual Exclusion

In the following program we indicate two parallel, cyclic processes (between the brackets **"parbegin"** and **"parend"**) that come into action after the surrounding universe has been introduced and initialized.

```
begin semaphore mutex;  mutex := 1;
  parbegin
      begin L1:  P(mutex); critical  section  1;  V (mutex);
        remainder of cycle 1;  go to L1
      end;
      begin  L2:  P mutex);  critical section  2;  V  (mutex);
        remainder of cycle 2;  go to L2
      end
  parend
end
```

As a result of the P- and V-operations on "mutex" the actions, marked as "critical sections" exclude each other mutually in time; the scheme given allows straightforward extension to more than two parallel processes,

the maximum value of mutex equals 1, the minimum value equals $-(n-1)$ if we have n parallel processes.

Critical sections are used always, and only for the purpose of unambiguous inspection and modification of the state variables (allocated in the surrounding universe) that describe the current state of the system (as far as needed for the regulation of the harmonious cooperation between the various processes).

Private Semaphores

Each sequential process has associated with it a number of private semaphores and no other process will ever perform a P-operation on them. The universe initializes them with the value equal to 0, their maximum value equals 1, and their minimum value equals -1.

Whenever a process reaches a stage where the permission for dynamic progress depends on current values of state variables, it follows the pattern:

P(mutex);
"inspection and modification of state variables including a conditional V(private semaphore)";
V(mutex);
P(private semaphore).

If the inspection learns that the process in question should continue, it performs the operation "V (private semaphore)"—the semaphore value then changes from 0 to 1—otherwise, this V-operation is skipped, leaving to the other processes the obligation to perform this V-operation at a suitable moment. The absence or presence of this obligation is reflected in the final values of the state variables upon leaving the critical section.

Whenever a process reaches a stage where as a result of its progress possibly one (or more) blocked processes should now get permission to continue, it follows the pattern:

P(mutex);
"modification and inspection of state variables including zero or more V-operations on private semaphores of other processes";
V(mutex).

By the introduction of suitable state variables and appropriate programming of the critical sections any strategy assigning peripherals, buffer areas, etc. can be implemented.

The amount of coding and reasoning can be greatly reduced by the observation that in the two complementary critical sections sketched above the same inspection can be performed by the introduction of the notion of "an unstable situation," such as a free reader and a process needing a reader. Whenever an unstable situation emerges it is removed (including one or more V-operations on private semaphores) in the very same critical section in which it has been created.

Proving the Harmonious Cooperation

The sequential processes in the system can all be regarded as cyclic processes in which a certain neutral point can be marked, the so-called "homing position," in which all processes are when the system is at rest.

When a cyclic process leaves its homing position "it accepts a task"; when the task has been performed and not earlier, the process returns to its homing position. Each cyclic process has a specific task processing power (e.g. the execution of a user program or unbuffering a portion of printer output, etc.).

The harmonious cooperation is mainly proved in roughly three stages.

(1) It is proved that although a process performing a task may in so doing generate a finite number of tasks for other processes, a single initial task cannot give rise to an infinite number of task generations. The proof is simple as processes can only generate tasks for processes at lower levels of the hierarchy so that circularity is excluded. (If a process needing a segment from the drum has generated a task for the segment controller, special precautions have been taken to ensure that the segment asked for remains in core at least until the requesting process has effectively accessed the segment concerned. Without this precaution finite tasks could be forced to generate an infinite number of tasks for the segment controller, and the system could get stuck in an unproductive page flutter.)

(2) It is proved that it is impossible that all processes have returned to their homing position while somewhere in the system there is still pending a generated but unaccepted task. (This is proved via instability of the situation just described.)

(3) It is proved that after the acceptance of an initial task all processes eventually will be (again) in their homing position. Each process blocked in the course of task execution relies on the other processes for removal of the barrier. Essentially, the proof in question is a demonstration of the absence of "circular waits": process P waiting for process Q waiting for process R waiting for process P. (Our usual term for the circular wait is "the Deadly Embrace.") In a more general society than our system this proof turned out to be a proof by induction (on the level of hierarchy, starting at the lowest level), as A. N. Habermann has shown in his doctoral thesis.