The structure of the THE-multiprogramming system

Introduction

Papers "reporting on timely research and development efforts" being explicitly asked for, I shall try to present a progress report on the multi-programming effort at the Department of Mathematics at the Technological University, Eindhoven, the Netherlands.

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 are faced with the problem of how to get the necessary experience. To solve this problem we have 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 onwards 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 precious experiences.

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

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

The one remark is that production speed is severely degraded if one works with half time people, who have other obligations as well. This is at least a factor 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 5 to 8 years, and are of Master's or Ph.D. level. A less qualified young man, originally included, found our activities beyond his mental grasp and left the group. I mention this explicitly, because at least in Holland, the intellectual level needed for system design is in general grossly underestimated. I am more than ever convinced that this type of work is just difficult and that every effort to do it with other than the best people is doomed to either failure or moderate success at enormous expenses.

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 2.5 mms., 27 bits; at present 32K.
2) drum of 512K words, 1024 words per track, rev.time 40 ms.
3) an indirect addressing mechanism very well suited for stack implementatior
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 1000 char/sec; 3 paper tape punches at 150 char/sec; 2 teleprinters, a plotter and 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
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 multi-access 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).

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 got more and more immodest as the work progressed.

<u>A progress report</u>

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

Our first major mistake has been 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 when, say, 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 part of our troubles were a direct consequence of our earlier ingenuity, i.e. the complexity of the situation into which the system could have manoeuvred itself. Had we paid attention to the pathology at an earlier stage in the design, our management rules would certainly have been less refined.

The second major mistake has been that we have conceived and programmed the major part of the system without giving more than scanty though to the problem of debugging it. For the fact that this mistake had no consequences - on the contrary ! one might argue as an afterthought - I must decline all credit. I feel more like having passed through the eye of the needle.....

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 is better than cure! - to try to prevent 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 it is possible to design a refined multiprogramming system in such a way that its logical soundness can be proved a priori and that its implementation admits exhaustive testing. The only errors that showed up during testing were trivial coding errors (occuring with a density of only one error per 500 instructions), each of them located within 10 minutes (classical) inspection at the machine and each of them correspondingly easy to remedy. At the moment of writing the testing is not yet completed, but the resulting system will be guaranteed to be flawless. When the system has been 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 occurences, for we shall have proved the correctness of the system with a rigour 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 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 as it originally came from. 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 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 co-operation 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 another 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 co-operation of a set of such sequential processes can be established by discrete reasoning; as a further consequence the whole harmonious society of co-operating 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.

On 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, 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. On the higher levels we find the activity of the different sequential processes, the actual processors having lost their 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 for the bookkeeping resulting from the automatic backing store, the dynamic relating of segments of information to pages of store. At this level our next abstraction has been achieved: in all higher levels identification of information takes place in terms of segments, the actual storage pages having 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 conservations 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 this next keyboard character: the actual printing is done on account of 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 for the benefit of the operator in the opening sentence of this conversation. 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! There lies the logical reason to introduce a separate sequential process for the console teleprinter, a reason that is reflected in its name "message interpreter". Above this level 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 a time", a restriction that is satisfied via mutual synchronization. At this level the next abstraction has been implemented: in all higher levels the actual console teleprinter has lost its identity. (If the message interpreter had been on the same level as 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 get 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. Therefore the message interpreter is of a level one 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 is 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 on higher levels, to be satisfied by mutual synchronization between them.

At level 4 we find the independent user programs, 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 this period of time the concepts have been born in terms of which we sketched the system in the previous section. Furthermore we learnt the art of reasoning by which we could deduce from our requirements the way in which the processes should influence each other as regards mutual synchronization so that these requirements were 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 learnt 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 behaviour satisfy all requirements.

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

In the verification stage we had, during short shots, the machine completely at our disposal, shots during which we worked with a virgin machine without any software aids for debugging. Starting at level 0 the system has been 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. Firstly they had to force the system into all different relevant states, secondly they had to verify that the system continued to react according to specification.

I shall not deny that the construction of these testing programmes has been a major intellectual effort: to convince oneself that one has not overlooked "a relevant state" and to convince oneself that the testing programmes generate them all is no simple matter. The encourageing thing is that (as far as we are aware!) 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 processor time was divided among them according to the rules.

This being established, sequential processes as such had been 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 synchronizing among the testing programs. At that 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 tried instead to make in a non-hierarchical construction the central processor directly react 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, drum speed and clock speed being outside our control, it is doubtful whether we would have had the means to generate them all.)

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 quite some time, but testing could proceed by restricting the number of segments so that they all 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 pience 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 relevant or not, cannot be decided as long as one regards the mechanism as a black box, in other words it has to follow from the internal structure of the mechanism to be tested. It seems the designer's responsibility to construct his mechanism in such a way - i.e. so highly structured - that at each stage of the testing procedure the number of relevant test cases is so small that he can try them all and that what is being tested is so perspicuous that it is clear that he has not overlooked a situation. I have presented a survey of our system because I think it a nice example of the form such a structure might take.

In my experience, I am sorry to say, industrial software makers tend to react to it with mixed feelings. On the one hand they are inclined to judge 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 Department and express the opinion that we could only do it this way thanks to the modest scope of the whole project. It is not my intention to underestimate the organizing ability needed for a much bigger job with ten or more times as many people, but I should like to venture 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, in particular 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.