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 XS (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 proce needing a reader. Whenever an unstable situation emerg it is removed (including one or more V-operations c private semaphores) in the very same critical section which it has been created.

Proving the Harmonious Cooperation

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

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

The harmonious cooperation is mainly proved in roughl three stages.

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

(2) It is proved that it is impossible that all process have returned to their homing position while somewhe in the system there is still pending a generated but una cepted task. (This is proved via instability of the situati just described.)

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

The Nucleus of a Multiprogramming System

PER BRINCH HANSEN
A/S Regnecentralen, Copenhagen, Denmark

This paper describes the philosophy and structure of a multiprogramming system that can be extended with a hierarchy of operating systems to suit diverse requirements of program scheduling and resource allocation. The system nucleus simulates an environment in which program execution and input/output are handled uniformly as parallel, cooperating processes. A fundamental set of primitives allows the dynamic creation and control of a hierarchy of processes as well as the communication among them.

KEY WORDS AND PHRASES: multiprogramming, operating systems, parallel processes, process concept, process communication, message buffering, process hierarchy, process creation, process removal
CR CATEGORIES: 4.30, 4.31, 4.32, 4.41

1. Introduction

The multiprogramming system developed by Regnecentralen for the RC 4000 computer is a general tool for the design of operating systems. It allows the dynamic creation of a hierarchy of processes in which diverse strategies of program scheduling and resource allocation can be implemented.

For the designer of advanced information systems, a vital requirement of any operating system is that it allow him to change the mode of operation it controls; otherwise his freedom of design can be seriously limited. Unfortunately, this is precisely what present operating systems do not allow. Most of them are based exclusively on a single mode of operation, such as batch processing, priority scheduling, real-time scheduling, or conversational access.

When the need arises, the user often finds it hopeless to modify an operating system that has made rigid assumptions in its basic design about a specific mode of operation. The alternative—to replace the original operating system with a new one—is in most computers a serious, if not impossible, matter because the rest of the software is intimately bound to the conventions required by the original system.

This unfortunate situation indicates that the main problem in the design of a multiprogramming system is not to define functions that satisfy specific operating needs, but rather to supply a system nucleus that can be extended with new operating systems in an orderly manner. This is the primary objective of the RC 4000 system.

In the following, the philosophy and structure of the RC 4000 multiprogramming system is explained. The discussion does not include details of implementation; size and performance are presented, however, to give an idea of the feasibility of this approach. The functional specifications of the multiprogramming system are described in detail in a report [1] available from Regnecentralen.

2. System Nucleus

Our basic attitude during the designing was to make no assumptions about the particular strategy needed to optimize a given type of installation, but to concentrate on the fundamental aspects of the control of an environment consisting of parallel, cooperating processes.

Our first task was to assign a precise meaning to the process concept, i.e. to introduce an unambiguous terminology defining what a process is and how it is implemented on the actual computer.

The next step was to select primitives for the synchronization and transfer of information among parallel processes.

Our final decisions concerned the rules for the dynamic creation, control, and removal of processes.

The purpose of the system nucleus is to implement these fundamental concepts: simulation of processes; communication among processes; creation, control, and removal of processes.

3. Processes

We distinguish between internal and external processes, roughly corresponding to program execution and input/output.

More precisely, an *internal process* is the execution of one or more interruptable programs in a given storage area. An internal process is identified by a unique process name. Thus other processes need not be aware of the actual location of an internal process in the store, but can refer to it by name.

A sharp distinction is made between the concepts program and internal process. A *program* is a collection of instructions describing a computational process, whereas an internal process is the execution of these instructions in a given storage area.

In connection with input/output, the system distinguishes between peripheral devices, documents, and external processes.

A *peripheral device* is an item of hardware connected to the data channel and identified by a device number. A *document* is a collection of data stored on a physical medium, such as a deck of punched cards, a printer form, a reel of magnetic tape, or a file on the backing store.

An *external process* is the input/output of a given document identified by a unique process name. This concept

implies that internal processes can refer to documents by name without knowing the actual devices on which they are mounted.

Multiprogramming and communication between internal and external processes are coordinated by the system nucleus—an interrupt response program with complete control of input/output, storage protection, and the interrupt system. We do not regard the system nucleus as an independent process, but rather as a software extension of the hardware structure, which makes the computer more attractive for multiprogramming. Its function is to implement our process concept and primitives that processes can invoke to create and control other processes and communicate with them.

So far we have described the multiprogramming system as a set of independent, parallel processes identified by names. The emphasis has been on a clear understanding of relationships among resources (store and peripherals), data (programs and documents), and processes (internal and external).

4. Process Communication

In a system of parallel, cooperating processes, mechanisms must be provided for the synchronization of two processes during a transfer of information.

Dijkstra has demonstrated that indivisible lock and unlock operations operating on binary semaphores are sufficient primitives from a logical point of view [3]. We have been forced to conclude, however, that the semaphore concept alone does not fulfill our requirements of safety and efficiency in a dynamic environment in which some processes may turn out to be black sheep and break the rules of the game.

Instead we have introduced message buffering within the system nucleus as the basic means of process communication. The system nucleus administers a common pool of *message buffers* and a *message queue* for each process.

The following primitives are available for the communition between internal processes:

send message (receiver, message, buffer),
wait message (sender, message, buffer),
send answer (result, answer, buffer),
wait answer (result, answer, buffer).

Send message copies a message into the first available buffer within the pool and delivers it in the queue of a named receiver. The receiver is activated if it is waiting for a message. The sender continues after being informed of the identity of the message buffer.

Wait message delays the requesting process until a message arrives in its queue. When the process is allowed to proceed, it is supplied with the name of the sender, the contents of the message, and the identity of the message buffer. The buffer is removed from the queue and made ready to transmit an answer.

Send answer copies an answer into a buffer in which a message has been received and delivers it in the queue of the original sender. The sender of the message is activated if it is waiting for the answer. The answering process continues immediately.

Wait answer delays the requesting process until an answer arrives in a given buffer. On arrival, the answer is copied into the process and the buffer is returned to the pool. The result specifies whether the answer is a response from another process or a dummy answer generated by the system nucleus in response to a message addressed to a nonexisting process.

The procedure wait message forces a process to serve its queue on a first-come, first-served basis. The system, however, also includes two primitives that enable a process to wait for the arrival of the next message or answer and serve its queue in any order.

This communication scheme has the following advantages.

The multiprogramming system is dynamic in the sense that processes can appear and disappear at any time. Therefore a process does not in general have a complete knowledge of the existence of other processes. This is reflected in the procedure wait message, which makes it possible for a process to be unaware of the existence of other processes until it receives messages from them.

On the other hand, once a communication has been established between two processes (i.e. by means of a message) they need a common identification of it in order to agree on when it is terminated (i.e. by means of an answer). Thus we can properly regard the selection of a buffer as the creation of an identification of a conversation. A happy consequence of this is that it enables two processes to exchange more than one message at a time.

We must be prepared for the occurrence of erroneous or malicious processes in the system (e.g. undebugged programs). This is tolerable only if the system nucleus ensures that no process can interfere with a conversation between two other processes. This is done by storing the identity of the sender and receiver in each buffer and checking it whenever a process attempts to send or wait for an answer in a given buffer.

Efficiency is obtained by the queueing of buffers, which enables a sending process to continue immediately after delivery of a message or an answer, regardless of whether or not the receiver is ready to process it.

To make the system dynamic, it is vital that a process can be removed at any time, even if it is engaged in one or more conversations. In this case, the system nucleus leaves all messages from the removed process undisturbed in the queues of other processes. When these processes answer them, the system nucleus returns the buffers to the common pool.

The reverse situation is also possible: during the removal of a process, the system nucleus finds unanswered messages sent to the process. These are returned as dummy answers to the senders.

The main drawback of message buffering is that it introduces yet another resource problem, since the common pool contains a finite number of buffers. If a process were

allowed to empty the pool by sending messages to ignorant processes, which do not respond with answers, further communication within the system would be blocked. Consequently a limit is set to the number of messages a process can send simultaneously. By doing this, and by allowing a process to transmit an answer in a received buffer, we have placed the entire risk of a conversation on the process that opens it.

5. External Processes

Originally the communication primitives were designed for the exchange of messages between internal processes. Later we also decided to use *send message* and *wait answer* for communication between internal and external processes.

For each kind of external process, the system nucleus contains a piece of code that interprets a message from an internal process and initiates input/output using a storage area specified in the message. When input/output is terminated by an interrupt, the nucleus generates an answer to the internal process with information about actual block size and possible error conditions. This is essentially the implementation of the external process concept.

We consider it to be an important aspect of the system that internal and external processes are handled uniformly as independent, self-contained processes. The difference between them is merely a matter of processing capability. A consequence of this is that any external process can be replaced by an internal process of the same name if more complex criteria of access and response become desirable.

External processes are created on request from internal processes. *Creation* is simply the assignment of a name to a particular peripheral device. To guarantee internal processes exclusive access to sequential documents, primitives are available for the *reservation* and *release* of external processes.

Typewriter consoles are the only external processes that can send messages to internal processes. The operator opens a conversation by pushing an interrupt key and typing the name of the internal receiver followed by a line of text.

A file on the backing store can be used as an external process by copying a description of the file from a catalog on the backing store into the system nucleus; following this, internal processes can initiate input/output by sending messages to the file process.

Real-time synchronization of internal processes is obtained by sending messages to a clock process. After the elapse of a time interval specified in the message, the clock returns an answer to the sending process.

In general, external processes can be used to obtain synchronization between internal processes and any signal from the external world. For example, an internal process may send a message to a watchdog process and receive an answer when a magnetic tape is mounted on a station. In response, the internal process can give the station a temporary name, identify the tape by reading its label, and rename the station accordingly.

6. Internal Processes

A final set of primitives in the system nucleus allows the creation, control, and removal of internal processes.

Internal processes are created on request from other internal processes. *Creation* involves the assignment of a name to a contiguous storage area selected by the parent process. The storage area must be within the parent's own area.

After creation, the parent process can load a program into the child process and *start* it. The child process now shares computing time with other active processes including the parent process.

On request from a parent process, the system nucleus waits for the completion of all input/output initiated by a child process and *stops* it. In the stopped state, the process can still receive messages and answers in its queue. These can be served when the process is restarted.

Finally, a parent process can *remove* a child process in order to assign its storage area to other processes.

According to our philosophy, processes should have complete freedom to choose their own strategy of program scheduling. The system nucleus only supplies the essential primitives for initiation and control of processes. Consequently, the concepts of program loading and swapping are not part of the nucleus. Time-sharing of a common storage area among child processes on a swapping basis is possible, however, because the system does not check whether internal processes overlap each other as long as they remain within the storage areas of their parents. Swapping from process A to process B can be implemented in a parent process as follows: stop(A); output(A); input(B); start(B).

7. Process Hierarchy

The idea of the system nucleus has been described as the simulation of an environment in which program execution and input/output are handled uniformly as parallel, cooperating processes. A fundamental set of primitives allows the dynamic creation and control of processes as well as communication among them.

For a given installation we still need, as part of the system, programs that control strategies of operator communication, program scheduling, and resource allocation; but it is essential for the orderly growth of the system that these *operating systems* be implemented as other programs. Since the difference between operating systems and production programs is one of jurisdiction only, this problem is solved by arranging the internal processes in a *hierarchy* in which parent processes have complete control over child processes.

After initial loading, the internal store contains the system nucleus and a basic operating system, S, which can create parallel processes, A, B, C, etc., on request from consoles. The processes can in turn create other processes, D, E, F, etc. Thus while S acts as a primitive operating system for A, B, and C, these in turn act as operating systems for their children, D, E, and F. This is illustrated by Figure 1, which shows a family tree of processes on the left

and the corresponding storage allocation on the right. This family tree of processes can be extended to any level, subject only to a limitation of the total number of processes.

In this multiprogramming system, all privileged functions are implemented in the system nucleus, which has no built-in strategy. Strategies can be introduced at the various higher levels, where each process has the power to control the scheduling and resource allocation of its children. The only rules enforced by the nucleus are the following: a process can only allocate a subset of its own resources (including storage and message buffers) to its children; a process can only start, stop, and remove its own children (including their descendants). After removal of a process, its resources are returned to the parent process.

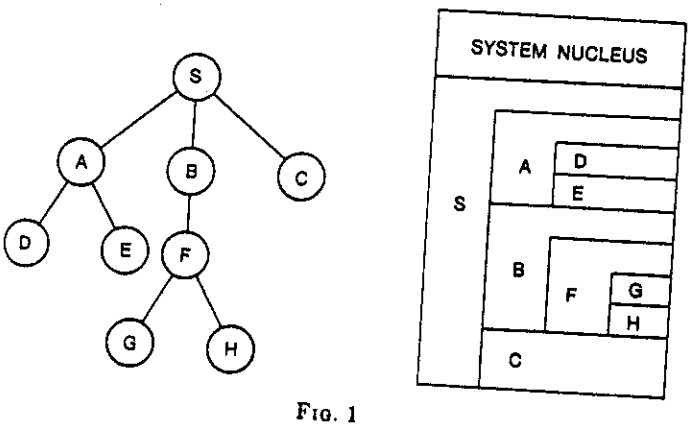

FIG. 1

Initially all system resources are owned by the basic operating system S. For details of process control and resource allocation, the reader should consult the manual of the system [1].

We emphasize that the only function of the family tree is to define the rules of process control and resource allocation. Computing time is shared by round-robin scheduling among active processes regardless of their position in the hierarchy, and each process can communicate with all other processes.

Regarding the future development of operating systems, the most important characteristics of the system can now be seen as the following.

1. New operating systems can be implemented as other programs without modification of the system nucleus. In this connection, we should mention that the ALGOL and FORTRAN languages for the RC 4000 contain facilities for calling the nucleus and initiating parallel processes. Thus it is possible to write operating systems in high-level languages.

2. Operating systems can be replaced dynamically, thus enabling an installation to switch among various modes of operation; several operating systems can, in fact, be active simultaneously.

3. Standard programs and user programs can be executed under different operating systems without modification, provided there is common agreement on the possible communication between parents and children.

8. Implementation

The RC 4000 is a 24-bit, binary computer with typical instruction execution times of 4 microseconds [2]. It permits practically unlimited expansion of the internal store and standardized connection of all kinds of peripherals. Multiprogramming is facilitated by program interruption, storage protection, and privileged instructions.

The present implementation of the system makes multiprogramming feasible with a minimum store of 16K–32K words backed by a fast drum or disk. The system nucleus includes external processes for a real-time clock, typewriters, paper tape input/output, line printer, magnetic tape, and files on the backing store. The size of the nucleus and the basic operating system is as follows:

	words
primitives	2400
code for external processes	1150
process descriptions and buffers	1250
system nucleus	4800
basic operating system	1400
	6200

The communication primitives are executed in the uninterruptable mode within the system nucleus. The execution times of these set a limit to the system's response to real-time events:

	msec
send message	0.6
wait answer	0.4
wait message	0.4
send answer	0.6

An analysis shows that the 2 milliseconds required by a complete conversation (the sum of the four primitives) are used as follows:

	percent
validity checking	25
process activation	45
message buffering	30

This distribution is so even that one cannot hope to increase the speed of the system by introducing additional, ad hoc machine instructions. The only realistic solution is to make the hardware faster.

The primitives for creation, start, stop, and removal of processes are implemented in an anonymous internal process within the system nucleus to avoid intolerably long periods in the uninterruptable mode. Typical execution times for these are:

	msec
create process	3
start process	26
stop process	4
remove process	30

(*Continued on page 250*)

The analysis presented here suggests that spatial domains are the primitive element of this particular graphic language. In this light, the common assumption that line segments are the primitives of many graphic languages may require revision.

RECEIVED JUNE, 1969; REVISED OCTOBER, 1969

REFERENCES

1. GROSS, MAURICE, AND NIVAT, MAURICE. A command language for visualization and articulated movements. In *Computer and Information Sciences II*, Julius T. Tou (Ed), Academic Press, New York, 1967.
2. NILSSON, NILS J. A mobile automaton: An application of artificial intelligence techniques. Proc. Int. Joint Conf. Artificial Intelligence, May 1969, Washington, D. C.
3. EASTMAN, CHARLES M. Explorations of the cognitive processes of design, Dep. of Comput. Sci., Carnegie-Mellon U., Feb. 1968, ARPA Rep. DDC No. AD671158, Clearinghouse, Springfield, VA 22151.
4. EASTMAN, CHARLES M. Cognitive processes and ill-defined problems: A case study from design, Proc. Int. Joint Conf. Artificial Intelligence, May 1969, Washington, D. C.
5. HOWDEN, W. E. The sofa problem. *Comput. J. 11*, 3 (Nov. 1968), 299-301.
6. SUTHERLAND, I. E. Sketchpad: a man-machine graphical communication system. Proc. AFIPS 1963 Spring Joint Comput. Conf., Vol. 23, Spartan Books, New York, pp. 329-346.
7. GRAY, J. C. Compound data structure for computer aided design: a survey, Proc. ACM 22nd Nat. Conf. 1967, Thompson Book Co., Washington, D. C., pp. 355-365.
8. THOMAS, E. M. GRASP—a graphic service program. Proc. ACM 22nd Nat. Conf., 1967, MDI Publications, Wayne, Pa., pp. 395-402.
9. ARMOUR, GORDON C., AND BUFFA, ELWOOD. A heuristic algorithm and simulation approach to relative location of facilities. *Man. Sci.* (Jan. 1963), 244-309.
10. LEE, R. B. AND MOORE, J. M. CORELAP—computerized relationship layout planning, *J. Indust. Eng., 18*, 3 (Mar. 1967) 195-200.
11. SIMPSON, M. G., ET AL. The planning of multi-story buildings: a systems analysis and simulation approach. Proc. European Meeting on Statistics, Econometrics and Management Science, Amsterdam, Sept. 1968.
12. BARKEN, ROBERT. A set of algorithms for automatically laying out hybrid integrated circuits. Internal working doc., Bell Telephone Lab., Holmdel, N. J., Aug. 1968.
13. NILSSON, N. J., AND RAPHAEL, B. Preliminary design of an intelligent robot. In *Computer and Information Sciences II*, Julius T. Tou (Ed.), Academic Press, New York, 1967.
14. ROSEN, C. A., AND NILSSON, N. J. Application of intelligent automata to reconnaisance. SRI Project 5953, Third Interim Report, Rome Air Develop. Center, Rome, N. Y., Dec. 1967.
15. FAIR, G. R., FLOWERDEW, ET AL. Note on the computer as an aid to the architect. *Comput. J. 9*, 1 (June 1966).
16. GRISWOLD, R., POAGE, J., AND POLONSKY, I. The SNOBOL4 programming language. Bell Telephone Lab., Holmdel, N. J., Aug., 1968.
17. MCCARTHY, JOHN, ET AL. *LISP1.6 Programmer's Manual.* MIT Press, Cambridge, Mass., 1965.
18. MORAN, THOMAS. Structuring three-dimensional space for computer manipulation. Dep. Comput. Sci. working paper, Carnegie-Mellon U., Pittsburgh, Pa., June, 1968.
19. MORAN, THOMAS. A model of a multi-lingual designer. In *Emerging Methods in Environmental Design and Planning*, G. Moore (Ed.), MIT Press, Cambridge, Mass. (in press).
20. WYLIE, C. ROMNEY, ET AL. Halftone perspective drawings by computer. Tech. Rep. 4-2, Comput. Sci. Dep., U. of Utah, Salt Lake City, Utah, Feb. 1968.

Hansen—*cont'd from page 241*

The excessive times for the start and removal of an internal process are due to the peculiar storage protection system of the RC 4000, which requires the setting of a protection key in every storage word of a process.

9. Conclusion

Ideas similar to those described here have been suggested by others [4–6]. We have presented our system because we feel that, taken as a whole, it represents a systematic and practical approach to the design of replaceable operating systems. As an inspiration to other designers, it is perhaps most important that it illustrates a sequence of design steps leading to a general system nucleus, namely, the definition of the process concept, the communication scheme, and the dynamic creation and structuring of processes.

We realize, of course, that a final evaluation of the system can only be made after it has been used to design a number of operating systems.

Acknowledgments. The design philosophy was developed by Jørn Jensen, Søren Lauesen, and the author. Leif Svalgaard participated in the implementation and testing of the final product.

Regarding fundamentals, we have benefited greatly from Dijkstra's analysis of cooperating sequential processes.

RECEIVED JULY, 1969; REVISED JANUARY, 1970

REFERENCES

1. *RC 4000 Software: Multiprogramming System.* P. Brinch Hansen (Ed.). A/S Regnecentralen, Copenhagen, 1969.
2. *RC 4000 Computer: Reference Manual.* P. Brinch Hansen (Ed.). A/S Regnecentralen, Copenhagen, 1969.
3. DIJKSTRA, E. W. Cooperating Sequential Processes. Math. Dep., Technological U., Eindhoven, Sept. 1965.
4. HARRISON, M. C., AND SCHWARTZ, J. T. SHARER, a time sharing system for the CDC 6600. *Comm. ACM 10*, (Oct. 1967), 659.
5. HUXTABLE, D. H. R., AND WARWICK, M. T. Dynamic supervisors—their design and construction. Proc. ACM Symp. on Operating System Principles, Gatlinburg, Tenn., Oct. 1-4, 1967.
6. WICHMANN, B. A. A modular operating system. Proc. IFIP Cong. 1968, North Holland Pub. Co., Amsterdam, p. C48.

matic problem which will strongly influence coming programming languages. Not only will conversational features be essential, there may even be a trend back from all too sophisticated language systems to the simple pointing with a light-pen. Pointing has always been one of the safest ways to convey information.

We come back to Wittgenstein and his principle of speaking clearly or not speaking at all. Since we know that it is the *computer* which we can make speak arbitrarily clearly, we possibly should try to let the computer speak more and more and to restrict the human user in the practical situation to point at YES or NO, or some more equally simple choices, while the computer talks. This may sound like science fiction today, but it could really be true that one day this will become the central application of pragmatics around the computer.

REFERENCES

1. PEIRCE, C. S. *Collected Papers*. Harvard Press, Cambridge, Mass. Vol. 1–6, 1931–1935.
   ——. *Philosophical Writings*. (J. Buchler, Ed.). Routledge and Kegan Paul, London, 1940; or Dover Publications, New York, 1955; 368 pp.
2. BOCHENSKI, I. M. *A History of Formal Logic*. U. of Notre Dame Press, Notre Dame, Indiana, 1961; pp. 99–100.
3. MORRIS, C. Foundations of the theory of signs. In *International Encyclopedia of Unified Science, Vol. 1, No. 2*, University of Chicago Press, Chicago, 1938.
4. ——. *Signs, Language, and Behavior*. G. Braziller, New York, 1955.
5. BOLTZMANN, L. *Vorlesungen ueber Gastheorie*. 1. Thei Paragraph 6. Mathematische Bedeutung der Groesse I J. A. Barth, Leipzig, 1895, pp. 38–42.
6. SHANNON, C. E. A mathematical theory of communicatior *Bell System Tech. J. 27* (1948), 379–433; 623–656.
7. KRAFT, V. *Der Wiener Kreis*. Springer Verlag, Vienna, 195(
8. WITTGENSTEIN, L. *Tractatus Logico-Philosophicus*. Firs Print in German, 1921; in English: Routledge and Kega Paul, London, 1922.
9. SCHLICK, M. *Gesammelte Aufsaetze*. Gerold and Co., Vienna 1938.
10. FEIGL, H. Logical empirism. In *Twentieth Century Philosoph*, (D. D. Runes, Ed.), Philosophical Library, New York, 1943 pp. 373–416.
11. CARNAP, R. *The Logical Syntax of Language*. First print i German, 1934; in English: Harcourt Brace and Co., New York, 1937.
12. ——. *Introduction to Semantics*. Harvard University Press Cambridge, Mass., 1942.
13. *Formal Language Description Languages* (T. B. Steel, Jr., Ed.). Proc. of the IFIP Working Conf., Vienna, 1964; North Holland, Amsterdam 1966.
14. GORN, S. Some basic terminology connected with mechanical languages and their processors. *Comm. ACM 4* (1961), 336–339.
15. ——. Mechanical pragmatics: a time motion study of a miniature mechanical linguistic system. *Comm. ACM 5* (1962), 576–589.
16. ——. Semiotic relationships in ambiguously stratified language systems. Presented at the Internat. Colloq. for Algebraic Linguistics and Automata Theory, Jerusalem, 1964.
17. MARTIN, R. M. *Towards a Systematic Pragmatics—Studies in Logics*. North Holland, Amsterdam, 1959.