BIT 13 (1973), 323–337

PROGRAM CONTROL OF OPERATING SYSTEMS

SØREN LAUESEN

Abstract.

The traditional job control language becomes superfluous if the existing programming languages are extended slightly. Such extensions also allow drivers and operating systems to be programmed entirely in high level languages. Ultimately, we may see machine independent operating systems. A framework is presented for an extendable operating system which allows a simple, uniform implementation of these language extensions.

1. Introduction.

This paper is concerned with what shall be called the *control language*, meaning the complete interface between the user and the operating system of a computer. This control language includes both statements of the job control language and statements of the programming language proper. The purpose of the paper is to show the advantages of an approach which eliminates the job control language by augmenting the ordinary programming languages with a few inter-process *communication primitives*.

In more detail, in present day systems the full interface between user and operating system has two parts. One part consists of *external control statements* of the job control language. They treat the compilers and object programs as procedures, and leave little or no possibility for the user program to modify the flow of control. The other part consists of the statements of the ordinary programming language for opening files and for calling input and output. In the approach proposed, the second part is extended so as to make the first part superfluous. This requires that the ordinary programming languages are extended with inter-process communication primitives and more complicated procedures for file definition, file creation, calling of translators, etc. External control statements as used to-day would then have to be expressed by an ordinary user program. Loops and conditions are then easily expressed, and we may think of the user program as controlling the operating system.

With an appropriate design of the extended programming language it will also become possible to handle peripheral devices in ways not anti-

Received September 15, 1972.

cipated by the designers of the operating system, and it will become possible to write operating systems entirely in a high-level language.

It seems feasible to introduce the communication primitives in all the major programming languages in a simple and uniform way, and so that mutual job security is maintained.

The advantages of this approach are, first, that the ordinary programming languages will become useful over a wider range of applications, second that the user control of the operating system will become more natural, and third that the control language will become as clearly defined as present day programming languages and will admit a similar degree of standardization.

Solutions to many of the technical problems are outlined in the paper. A set of communication primitives and a set of basic operating system features are proposed in Section 5. One major technical problem remains as discussed in Section 5.5: The design of a strategy for working store allocation which is efficient, flexible, and allows the sharing needed for the communication primitives.

The incentive for this paper was the Nordic Working Conference on Basic Software, held in Copenhagen on October 1971 [3]. A main theme of this conference was the possibility for constructing a machine independent control language for operating systems. Starting from the existing major operating systems it seems tempting to propose a common job control language, including conditions, loops, and procedures (macros), and with an Algol-like appearance. In the author's opinion this approach would hide the fundamental problems of the field (besides adding to the Tower of Babel). I therefore submitted the present proposal, in a preliminary form, to the conference. Later I have found that the CODASYL Data Base Task Group follows a similar approach in another field [1, 2].

2. Interface between user and operating system.

The *control language* is defined as the complete interface between user and operating system. The commands of a control language may be classified roughly as follows:

External control commands specify data files, calls of translators, resources for the job step, etc. They are in many systems called "control cards" and they may be the only control commands of which the ordinary user is aware. The external control commands are interpreted by the operating system and form the external control language (sometimes called the command language or the job control language).

Internal input/output commands specify block transfers, etc. and are invoked by the translators and the input/output procedures of the user programs. The information is communicated in binary form as various supervisor calls behind the back of the ordinary user.

Internal declarative commands specify things like open file, reserve working store, end job. They appear much like the input/output commands, but are less frequent and involve a lot of housekeeping.

The internal input/output commands and declarative commands form the *internal control language.*

In most systems some external control commands do not have a corresponding internal control command. For instance in OS/360 it is necessary to use a DD control card to get access to a file. In early versions, even the connection between a logical unit and a physical dataset had to be stated in the DD card [8].

3. Control languages viewed as languages.

3.1. *Classical programming languages.*

Let us try to see how the classical programming languages Algol, Cobol, and Fortran perceive the computer in order that we can analyze the difference between for instance Algol 60 and a job control language.

Obviously, Algol 60 perceives the computer as a large working store plus a control and arithmetic unit. The working store contains the program and all the variables. The *declarations* of the language specify the structuring of the store. By means of *operators* we can perform simple operations on the structure, by means of procedures we can form more complicated operators.

In Cobol the computer is further viewed as equipped with a set of sequential files structured as records. The structuring and the files are specified statically in the environment and data divisions.

Fortran, as far as files are concerned, takes a position between Algol and Cobol.

3.2. *External control languages.*

Unfortunately, a modern computer contains many things which do not fit these simple pictures: A very *varied set of peripheral devices*, each with its own unique properties. A *dynamic set of files* (usually on disc) which may be structured in many ways. A set of *jobs* executed in parallel and more or less dependent of each other. (In particular, I do not mention the interrupt system, because I believe that interrupts should be hidden in the lowest level possible [22]).

It seems that job control languages are invented in order to utilize these possibilities without changing the classical programming languages. A job control language may be described crudely as a poor programming language working mainly with the data types "files" and "jobs" [6, 7]. Usually, not even all existing devices are included, as software cannot keep up with hardware development. The execution is normally strictly sequential, job step by job step, without the conditions and loops of a higher language.

The most peculiar thing is that all operators (commands) of the job control language are very complicated, for instance working with entire files. In Algol 60, it would correspond to omitting simple variables and allowing only operators like "matrixmult" and "matrixadd". Such a language is very useful for certain limited problem fields, and one of the troubles with the job control languages is exactly that they only work for a very limited problem field, which just happens to have been dominating for a long time: fast handling of many independent jobs.

In light of this, most new programming languages are very frustrating as they mainly present new ways of structuring the working store (e.g. Algol 68, Simula, APL, Snobol, Basic).

Furthermore, papers about general operating system principles tend to stress the concept of an external control language [6, 7, 9, 21].

3.3. *Internal control language.*

If instead we try to think of the troubles such as the inability of the classical programming languages to handle the new data types (devices, files, jobs), another solution turns up: Make a corresponding set of new basic data types and introduce them in all languages. Define declarations for them. Introduce the necessary basic operators working on them. Finally, introduce a suitable set of standard functions covering the frequently used composite functions now expressed by means of control cards. If the basic data types and operators are chosen properly, the standard functions may be expressed entirely in terms of basic operators.

The basic operators may be classified as internal declarative commands (which specify the structuring of devices, files, etc.) and internal input/output commands (which perform simple operations on the structures).

When using an internal control language, the sophisticated user has full freedom to handle his part of the computer. Security is still maintained as the operating system prevents the user from destroying other parts of the system. The simple-minded user need not worry about all these possibilities, because he may use the standard functions solely.

4. Using the internal control language.

When a user logs in on the system, he will have to state his user name and password (project number). This is all of what remains of the old external control language. Next, the user communicates through his initial programming language, for instance Algol 60. The compiler will now read a program, compile it, execute it, read the next program, and so on. A program may correspond to a single control command or to an entire algorithm.

If the user is so simple-minded, that he believes the computer just has a large working store for him, he need know nothing more about control commands. If the user wishes to use files, generate subjobs, or switch to another programming language, he can express this as statements in his current programming language.

Some languages are more suitable as initial languages than others. For instance, if I were using a non-conversational Cobol compiler, I would choose some other language as the initial programming language (a language like Basic perhaps), and then invoke the Cobol compiler later.

5. Proposal for a general operating system framework.

In this section I will outline an operating system suitable for introduction of a simple internal control language. Since many unimplemented functions and all unimplemented strategies may be executed manually by the operator (but hopefully slower than an automatic system), the operating system may be implemented gradually.

5.1. *Process concept, standard communication.*

The system may be thought of as a set of parallel processes, i.e. a set of programs executed simultaneously. Some of these processes are jobs in the normal sense, others are drivers communicating with peripheral devices and receiving device interrupts, still others are called operating systems because they control job processes.

These processes communicate with each other by means of a standard set of communication procedures. Several choices of such sets exist [4, 8, 11, 13, 20, 23]. My favourite set resembles the proposal of [4], and employs just two procedures: *send a message* to a specified queue and *wait for a message* in a specified queue. These two procedures correspond closely to the semaphore operations of [11], but they associate a queue of messages to each semaphore. This has the advantage of resolving the problem of [15] and enables a safe communication between erroneous processes. The procedures have been used successfully in the Boss 2 operating system for RC 4000 [18, 19].

In this communication the queues exist rather independently of the processes, which distinguishes the system from the message communication system in [13, 14]. In general only a subset of the processes may use a certain queue. A message is a part of the sender's working store area (often containing a buffer) which is "disconnected" at the moment of "send". When a process receives a message as a result of "wait", it is treated as an extension of the receiver's working store area.

The message will typically contain an operation code (input, output, backspace, etc.) and the buffer involved in the data transfer between sender and receiver. In most cases the message is sent to the receiver and later back again as an answer, and then the message must specify the answer queue.

This proposal for communication is used in the sequel, but other forms of standard communication might be substituted—except in the examples of section 5.8.

The communication procedures and the code for the time multiplexing between the parallel processes are part of the *monitor* (supervisor).

5.2. *Operating systems and jobs.*

Each process possesses a set of resources as follows:
An area of the working store.
A set of queues for which the process is allowed to wait.
A set of queues to which the process may send.
A maximum fraction of the available CPU-time.
A set of peripheral devices (this set is empty except for drivers).

If the process attempts to exceed its resources, it will get a warning signal from the corresponding procedure, showing that the operation was not carried out, but it will not be aborted automatically or otherwise influenced.

Every process may now use a subset of its resources to create one or more new processes—*child processes* with the creator as *parent*. Later, the parent may increase or decrease the resources of a child, possibly as a consequence of messages received from the child. Eventually the parent may remove a child entirely, which causes the resources to be returned to the parent.

A problem arises with the resources sent as messages (part of the working store of the child) to another receiver process. They may either be returned to the parent's resources (via a special queue) when later answered by the receiver process, or they may be forced back to the parent when the child is removed (the solution of [5]). The original solution to the problem in [13] corresponded to the first possibility, but with the special queue omitted; as a consequence, the parent could not keep track of his resources except by means of "busy waiting" (i.e. asking with regular intervals whether the situation has changed).

As a child can only have a subset of the parent's resources, the child can do no more harm than the parent, so that child creation in principle is allowed for all processes.

The parent process also determines the initial contents of the child's working store, i.e. the initial program of the child. If the initial program is a compiler or an interpreter for a job control language, the parent is an operating system and the child is a job.

The procedures for creation and removal of children and for resource delegation are the remaining part of the monitor.

A child process may again in principle create its own children, but normal jobs do not do this of course. However, the possibility allows debugging of new operating systems and drivers under the control of existing operating systems. Initially, the computer contains one process only: a simple operating system which possesses all resources and which is able to create jobs and drivers according to commands typed in by the main operator.

5.3. *Job execution.*

When the user logs in, he will first talk to an operating system, which will read his identification, look it up in a user catalog and determine the initial program (programming language) of the job. A certain set of maximum resources (resource claims) is also specified by the user or by default in the user catalog. This information is needed for the later dynamic resource allocation in order to avoid Deadly Embrace [12]. Essentially the same steps are followed when the user submits a batch job instead of working on-line.

At a suitable moment, the operating system will create a child process (the job) with the specified initial program. The job is informed of a queue to use for obtaining the first stream of input, a queue to use for printing messages, and a queue to use for communicating with the operating system. These three parameters are sufficient for the job to proceed. For an on-line run, the input stream corresponds to lines (or characters) typed in at the terminal and the print stream corresponds to output to the terminal.

Presumably, the job will soon need access to some file on the disc. It will then send a message to the operating system asking for "opening" of the file in question. The operating system will check to see that the request is legal (this involves lookup in file catalogs) and at a suitable moment the operating system creates a child process (a driver) to handle the file. The driver will be allowed to receive messages from a certain queue and the job will be allowed to send messages to this queue. When the job receives the answer from the operating system (specifying the queue, etc.), it can go ahead communicating with the driver in order to access the blocks of data in the file. The splitting of the blocks into characters or logical records is done entirely inside the job in most cases.

Creation or removal of files on the disc is also requested by means of messages to the operating system. In general all internal declarative commands are issued as messages to the parent, as only the parent may change the resources of the child. Further, these operations involve strategic decisions concerning all jobs of the parent. Once the connection is established between job and driver, the internal input/output commands may proceed rather fast, without bothering the parent.

Notice that the declarative message specifies the physical file (or volume) and not some logical unit invariant from run to run. This is essential for the omission of an external control language, but it also allows the program to handle things like incarnations of files automatically by means of its own algorithm, as is needed in business applications for instance.

5.4. *Drivers*.

A real driver is a process which possesses a peripheral device with which it communicates as prescribed by hardware (input/output instructions and interrupts, for instance as in [22]). Towards jobs and operating systems it exhibits the standard interface of message communication.

Any process may work as a pseudo-driver: Towards jobs and operating systems it exhibits the standard interface, but instead of executing input/output instructions, it communicates with a real driver. Since complexity (e.g. spooling) may be added as pseudo-drivers—possibly specifically for a particular application—all real drivers should be kept very primitive. In this way, the larger working store necessary for the complexity may be allocated by the operating system, which usually is superior to a low level store allocation done by the monitor.

On some computers the interrupt and input/output system is sufficiently simple, so that an ordinary job may be allowed to perform some input/output directly without a driver. This may be possible for devices which are not shared between jobs—magnetic tape stations, process control devices, etc.

The pseudo-driver principle enables the following handling of several files on the same disc: The disc has a real driver which requires absolute physical disc addresses in the messages. When a file is opened, the operating system creates a corresponding pseudo-driver which transforms the logical segment numbers from the job into absolute segment numbers (after appropriate checking) and passes the modified message on to the real driver. In such cases it is advantageous to have reentrant driver processes, as several files are open simultaneously.

The checking and error recovery needed after input/output-transfers may be done partly in the drivers, partly in checking procedures inside the job. In the RC 4000 system [16] a user specified checking procedure is associated with each open file in Algol and Fortran programs. The checking procedure may be called as a side-effect of all wait operations, especially those which complete input/output-transfers.

5.5. *Working store optimizing.*

Until now we have ignored the problem of storage allocation occurring when all jobs cannot be kept in the working store at the same time. Two main techniques are in use: 1) *Swapping* and roll-in/roll-out, where a job occupies a contiguous area of the working store which may be saved (swapped) temporarily on drum or disc. 2) *Paging* or virtual storage, where only some pages of the job's virtual storage area need be in the working store.

In both cases we are faced with a buffer problem, a detection problem, and a delegation problem.

The buffer problem occurs when it is decided to save a part of the working store on drum or disc and the part contains input/output buffers. In this case the part is *stopped* which means that the save is delayed until the buffer transfers are completed and further transfers to that part are prevented. With a paging scheme the page containing the buffer can easily stay in the working store during the stop, but with a swapping scheme the entire job area has to stay, which is unacceptable for slow buffer transfers. In the RC 4000 system [13, 14] a parent process may ask the monitor to execute such a stop of a child process, in this way preparing for a swap. This stopping causes severe problems for the drivers and various solutions are employed. For instance, drivers for slow devices are prepared to halt in the middle of a buffer transfer leaving it to the child to repeat the transfer when it is later swapped back to the working store. When two jobs share the same place in the working store—one swapped out while the other runs—they cannot communicate by means of buffers at all.

The detection problem occurs when a job initiates a wait for a time consuming operation. This will be an excellent moment to swap the job or—with a paging scheme—to give low priority to all pages of the job. As the system is outlined above, the operating system knows in many cases when the job sends a message specifying a time consuming operation (e.g. an internal declarative message), but the moment when the job starts waiting for the answer remains hidden. In RC 4000 we have sometimes doubled the communication for this purpose. For instance, we have a message specifying "mount a magnetic tape and let me continue" and another one specifying "mount a magnetic tape and answer when ready". The detection problem is circumvented in systems where a part of the working store (a job or a page) is stopped and saved when the part has been unused for a certain time (the working set model [10]). This technique is inefficient in detecting that a job awaits a slow operation—which is to be expected as the technique discards essential information from the communication primitives. When the time-consuming operation is completed, similar detection troubles occur.

The delegation problem concerns the ability of a parent to influence the storage allocation strategy of its children. Swapping schemes may allow a parent to apply its own swapping strategy to the children, but it seems that paging schemes must be implemented at the lowest level possible: in the heart of the monitor, which makes experimentation with the strategy a dangerous thing.

5.6. *Gradual implementation of the operating system.*

The set of declarative commands to the operating system will be rather comprehensive, but it is possible to make a design so that the operating system can handle all "unknown" commands in the same way: the message is displayed to the operator in a standard format containing the job identification. The operator then decides whether the message is legal, and at a suitable moment he performs the operation by means of the following basic commands implemented in the operating system from the beginning:

Answer a specified message.

Create a process with a specified program (driver or job).

Allow a process to use some queues.

Remove a process.

As the system is improved, the operating system may take over the handling of more and more messages.

As an example, consider an operating system which ignores the request "mount mag tape 5001", but just displays it to the operator. The operator is now responsible for the legality of the job using the tape. He mounts the tape, asks the operating system to create a driver for the tape, allow the job to communicate with the driver queue, and send the answer to the job. A more advanced operating system takes care of selecting a free station, detecting the tape by means of the label as soon as it is mounted, and checking the access rights of the job.

This form of interface was designed by the author in 1969 for RC 4000 —several months after the completion of the monitor. The interface has since then allowed jobs to run without change under very different operating systems [18].

5.7. *The internal control language.*

In order to utilize the interface above, all major programming languages should be extended with a set of basic commands and data types.

In case of the message communication proposed, data types describing messages and jobs seem necessary. At least in Algol 60 the working store areas holding the message buffers and child processes need special treatment in the stack on block exit, as one must ensure that no modifications by parallel processes take place after completion of the block exit. The queues need not be new data types (although this may be convenient) as they may be represented by integers in the same way as in machine language; the monitor will be able to check them anyway.

In a language like Algol 60, which does not have means for composing data types into records, it will be advantageous to introduce a more complex data type like a multi-buffered file – as files are so common— and then allow access to the constituents of this data type by means of special operators or procedures. Some of the constituents will be messages and jobs, so that no generality is lost. This approach was followed in Algol 60 and Fortran for RC 4000 [16].

Basic operations for message sending and waiting, and process creation and removal are necessary. Standards for message formats must be adopted in case transferability of the programs is wanted. Although this will be a difficult task, I feel that only a common approach to the basic principles may avoid a hodgepodge like PL/1.

The final step is to implement procedures for the composite commands needed in present external command languages, but this is comparable to implementing procedures for matrix operations, Bessel-functions, sorting, etc. In fact the composite commands might be programmed in one of the extended programming languages. A consequence of the approach is that all utility programs and compilers should be available as procedures in all the languages.

5.8. *Examples of message communication.*

A multi-buffer scheme for input/output is established by the user program when it sends two or more messages asking for input/output before it awaits the first answer. The driver need not care about the difference. The scheme is exactly the elegant producer-consumer algorithm of Dijkstra [11], but with the pointer updating taken care of by the queues.

The method can be modified to the case where for instance 3 files share 4 buffers in an optimal way. Notice, that everything is organized by the user program (possibly by means of library procedures), so that the operating system need not care.

A multi-access system for terminals (resembling QTAM of OS/360) is established when the user program sends one message to each of the terminals involved and specifies the same answer queue for all of them. Waiting for the first answer in the queue then allows the job to handle the terminals in a sequential manner.

Critical regions between processes may be established by means of a common queue (corresponding to the binary semaphore of Dijkstra). The variables involved in the critical region are sent as a message to the queue and may be waited for by any of the processes involved, but only one process at a time may have access to the message.

6. Comparison with existing systems.

The system proposed above has the following characteristics:

1. A standard form of communication between all processes.
2. Declarative functions and resource control handled by exchangeable operating systems and not in the monitor (permanent nucleus).
3. Dynamic creation and removal of processes.
4. Dynamic definition of physical files controlled by user's program.
5. Extendable interface to the operating system with possibility for gradual implementation.
6. Internal control commands comprising the entire interface to operating system and drivers.
7. Composite control commands in all programming languages making each of them suitable as a control language.
8. Possibility for pseudo-drivers programmed entirely in high level languages.

9. Possibility for operating systems programmed entirely in high level languages.

The system is much influenced by the RC 4000 system [5, 13, 14, 16, 17, 18, 19], which however falls short of the goal in the following four essential points:

1) The message communication of RC 4000 does not allow the simple solutions of the examples in section 5.8. Nor is it possible to introduce pseudo-drivers in the general case.

2) In RC 4000 many of the declarative functions are built into the monitor (for instance all functions concerning backing store) and all systems programs utilize these functions. As a consequence, the operating system cannot take over these functions without changes in the present programs, because communication with the operating system would be needed instead of calls of monitor functions. This has been a severe restriction as the monitor had to be modified a lot before an advanced operating system could be implemented [5, 18].

3) The monitor function for process removal was incomplete as it could not guarantee the return of all resources to the parent. This has been remedied as explained in [5].

4) The Algol and Fortran languages for RC 4000 contain all the basic operations of an internal control language, and in fact operating systems and sophisticated input/output strategies have been implemented in Algol. But the composite procedures corresponding to normal job control commands have not been written, so that the ordinary user has to use a simple, interpretive language for job control [17]. The interpreter is loaded as the initial program of the job according to the proposal above.

Most other dialects of Algol 60, Fortran, and Cobol are very far from the requirements above. It seems that PL/1, with multi-tasking and WAIT-statements, is close to the goal, although very much depends on the operating system. Under OS/360 a standard communication does not exist, resources cannot be controlled for subtasks by the PL/1 program, an extendable interface does not exist, and PL/1 cannot be used as a control language in the general case.

Burroughs's Extended Algol is comparable to PL/1 in these respects. The ZIP-statement enables the language to be used as a control language.

In most systems, the interface between jobs and operating systems suffers from the same flaws as the RC 4000 system. Too much is put into the central system without allowing another (experimental) operating system to intervene. For instance, in IDA [23] the file system is part of the supervisor and only single user devices may be handled freely.

Another general flaw is that the communication between systems parts is heterogeneous. For instance in OS/360 a lot of special supervisor calls exist and two basic process communication methods are in use: WAIT/POST and ENQ/DEQ [8]. The communication method in IDA is very interesting, but it employs busy waiting (i.e. inspection of conditions at regular intervals even though no changes have occurred).

An interesting system with a sound communication method and without an external job control language is described in [4]. Unfortunately the system is not suited to support other languages than the language constructed for the purpose.

Acknowledgements.

The ideas in this paper have originated during years of inspiring collaboration with Peter Lindblad Andersen, Jørn Jensen, Klavs Landberg, and Per Mondrup. Finally I would like to thank Peter Naur for his criticism which forced me to sharpen my thoughts.

REFERENCES

1. *The debate on data base management*, EDP Analyser, March 1972.
2. *Report of the Codasyl Base Task Group*, April 1971.
3. Nils Andersen, and Niels Gellert, (eds.), *Nordic Working Conference on Basic Software*, Dansk Selskab for Datalogi, Copenhagen, 1972.
4. P. Lindblad Andersen, J. Jensen, P. Jensen, and J. Steensgaard-Madsen, *Grok*, Datalogisk Institut, Copenhagen, 1972.
5. P. Lindblad Andersen, *Monitor 3*, RCSL No. 31–D109, Regnecentralen, Copenhagen, 1972.
6. D. W. Barron and I. R. Jackson, *The evolution of job control languages*, Software-Practice and experience, Vol. 2, p 143–164 (1972).
7. J. Bubenko and T. Ohlin, *Introduktion till operativsystem, Del 2*, Studentlitteratur Lund 1971.
8. W. A. Clark, G. H. Mealy, and B. I. Witt, *The functional structure of* OS/360, IBM Systems Journal, no. 1, 1966.
9. G. Cuttle and P. B. Robinson, (eds.), *Executive programs and operating systems*, Mac Donald, London 1970.
10. P. J. Denning, *Thrashing: Its causes and prevention*, AFIPS 1968 FJCC, Vol. 33, pp. 915–922.
11. E. W. Dijkstra, *Cooperating sequential processes*, In "Programming Languages", F. Genuys (ed), Academic Press, New York, 1968, pp. 43–112.
12. A. N. Habermann, *Prevention of system deadlocks*, Comm. ACM 12,7 (July 1969), pp. 373–377,385.
13. P. Brinch Hansen, *The nucleus of a multiprogramming system*, Comm. ACM 13,4 (April 1970), pp. 238–250.

14. P. Brinch Hansen, RC 4000 *Software, multiprogramming system*, RCSL No: 55–D140. Regnecentralen, Copenhagen, 1971.
15. D. E. Knuth, *Additional comments on a problem in concurrent programming control*, Comm. ACM 9,5 (May 1966), pp. 321–322.
16. S. Lauesen, RC 4000 *Software, Algol 5*, RCSL No: 55–D141. Regnecentralen, Copenhagen, 1970.
17. S. Lauesen, RC 4000 *Software, file processor*, RCSL No: 55–D21. Regnecentralen, Copenhagen, 1969.
18. S. Lauesen, *Boss 2, User's Manual*, RCSL No: 31–D211, Regnecentralen, Copenhagen, 1972.
19. S. Lauesen, *Boss 2, Installation and Maintenance*, RSCL No: 31–D191 Regnecentralen, Copenhagen 1972.
20. E. I. Organick, and M. J. Spier, *The multics interprocess communication facility*, Proc. Second Symp. on Oper. Syst. Princ., ACM, New York, 1971, pp. 83–91.
21. H. R. Wiehle, C. Seegmüller, W. Urich, and F. Peischl, *A monitor system for high-speed computers*, Elektron. Rechenanl. 6(1964), H.3, pp. 119–125.
22. N. Wirth, *On multiprogramming, machine coding, and computer organization*, Comm. ACM 12,9 (Sept. 1969), p. 489–498.
23. R. Stockton Gaines, *An operating system based on the concept of a supervisory computer* Comm. ACM 15,3 (March 1972), pp. 150–156.

NORDISK BROWN BOVERI, COPENHAGEN, DENMARK