Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

MC SYLLABUS 32

COLLOQUIUM BEDRIJFSSYSTEMEN

MATHEMATISCH CENTRUM AMSTERDAM 1976

AMS(MOS) Subject classification scheme (1970): 68-00, 68A10

ACM -Computing Reviews- categories: 4.30, 4.32, 4.21, 4.35

ISBN 90 6196 137 8

INHOUD

Voorwoord ix

I. A SURVEY OF OPERATING SYSTEMS by A.S. TANENBAUM

0. Introduction 1
1. Operating system classification and capabilities. 1
2. Processes 5
3. Operating system structure. 9
4. Resource management 12
5. Protection 16
6. Performance monitoring and evaluation 21
References 23

II. VIRTUALITY by H.J. BOOM

1. Virtual storage 25
2. Virtual CPU's 27
3. Protection. 28
4. Virtual input and output. 29
5. Virtual machines 30
6. Virtual time. 30
7. Virtual paging. 31
8. Resource maps 32
9. High-level virtuality 34
10. Implementation techniques 35
11. Virtual data. 36
12. Applications. 38
13. Files and storage 38
14. Protection. 39
15. Conclusion. 41
Reference. 41

III. ENIGE ASPECTEN VAN HET BURROUGHS B6700 SYSTEEM door H. ROUMEN

0. Inleiding 43
1. Machine-architectuur. 44

2. Geheugenbeheer 49
3. Job en task scheduling 53
4. Protectie 56
5. Datacommunicatie 57
6. BEA en ESPOL 58
Literatuur. 59

IV. VAN BLOKKENDOOS TOT BEDRIJFSSYSTEEM door P. KLINT

1. Inleiding. 61
2. Implementatiehulpmiddelen. 63
3. Betrouwbaarheid. 72
4. Metingen 79
5. Slotopmerking. 81
Literatuur. 82

V. MULTIPROCESSING IN DE CYBER COMPUTERS door H. BARREVELD

1. Inleiding. 83
2. De computer. 83
3. Het besturingssysteem. 93

VI. SOME ASPECTS OF THE EFFICIENCY OF SYSTEM IMPLEMENTATION LANGUAGES by B.A. WICHMANN

0. Introduction 101
1. Efficiency 101
2. How high level? 102
3. Data structures 106
4. Conclusions. 107
5. Some examples. 108
References. 111

VII. JOB CONTROL LANGUAGES door L.G.L.T. MEERTENS

0. Inleiding. 113
1. JCL als programmeertaal. 114
2. Waarom een aparte JCL? 115
3. Het profiel van een hogere JCL 117
4. Stromen. 118
5. Processen. 121

6. Localiteit 122
7. Editing. 124
8. Synchronisatie en foutbehandeling. 125
9. Conclusie. 126
Literatuur. 127

VIII. OVER DE IMPLEMENTATIE VAN SYNCHRONISATIECONCEPTEN door H.J.M. GOEMAN

0. Inleiding. 129
1. De kritieke sectie 130
2. De actieve arbiter 134
3. Andere synchronisatieconcepten 142
Literatuur. 146

VOORWOORD

In deze Syllabus zijn de voordrachten gebundeld, gehouden in het academisch jaar 1975/1976 in het door de Afdeling Informatica van het Mathematisch Centrum georganiseerde Colloquium Bedrijfssystemen.

Gaarne betuig ik mijn dank aan de sprekers in het Colloquium, in het bijzonder de niet aan de Afdeling verbonden gastsprekers, en aan L.J.M. Geurts voor zijn aandeel in de organisatie van het Colloquium en de verzorging van de Syllabus.

J.W. de Bakker

MATHEMATICAL CENTRE SYLLABUS 32, 1976, I, 1-24

A SURVEY OF OPERATING SYSTEMS

A.S. TANENBAUM
Vrije Universiteit, Amsterdam

0. INTRODUCTION

Like the 3 blind men exploring the elephant, different people viewing the same operating system can come to radically different conclusions as to what the "true nature" of the beast is. Some people define "operating system" as the totality of software delivered by their computer manufacturer, including the COBOL compiler payroll program and tape sort routine. Other, more discriminating, folks view an operating system as a collection of programs whose function is to manage the computer's resources (e.g. CPU, primary memory, peripherals, disks, etc.) in order to maximize performance. Still other people regard the operating system as a partial interpreter whose task is to hide the unpleasant characteristics of the hardware from the programmer. In this view, the operating system's function is to provide the user with an "extended" or "virtual" machine, one which is more convenient to use than the original.

Prepared with the knowledge that no two people seem to agree on what an operating system (henceforth OS) is, it should come as no shock to the reader that there is no generally accepted breakdown of the subject into subdisciplines. In this article we will examine 6 topics that this author considers the essence of the subject: functional capabilities, processes, structure, resource management, protection, and evaluation. The subjects treated here are all areas of current research.

1. OPERATING SYSTEM CLASSIFICATION AND CAPABILITIES

1.1. *Taxonomy of operating systems*

The simplest OS type is the monoprogramming system. In this system the operator manually enters a job. When the job is finished, the operator man-

ually enters the next one. Many stand alone minicomputers use a monoprogrammed OS.

To eliminate wasted time between jobs, the batch system was invented. In this system a collection of jobs are strung together on some input medium (magnetic tape, card decks, or a disk file). The batch system simply runs them in sequence without operator intervention. Each job waits until its predecessor has terminated before starting.

Most programs must ocassionally pause to wait for i/o to complete. In a batch system the CPU is idle during i/o wait time. In a commercial data processing environment, i/o wait time is often 90% or more. To increase CPU utilization, multiprogramming systems were invented. In a multiprogramming system, several programs are kept in primary memory simultaneously. When one program is blocked, waiting for i/o to complete, the CPU can be given to another program. If the primary memory is sufficiently large, there will nearly always be an unlocked program around, and CPU utilization can be raised to close to 100%.

Multiprogramming, although a valuable technique, brings with it a host of complications. For example if there is insufficient primary memory, sometimes all programs will be blocked, and CPU time will be wasted. If primary memory is added to a system already running at 100% CPU utilization, the added memory is unnecessary, hence wasted. Achieving a balance of CPU, memory and other resources in the face of a dynamically varying load is not easy. Nevertheless, most OS's nowadays, except for some mini's, are multiprogrammed, and we will concentrate our study on these.

If some of the programs running under a multiprogramming system can interact with a human being at a remote terminal, the system is called a timesharing system. Timesharing introduces the additional complication of the need to swap programs waiting for human input out of primary memory.

Some multiprogramming systems have remote terminals consisting of laboratory equipment or industrial process control devices instead of teletype or CRT terminals. These devices send data to the connected program for storage or processing. If there is no buffering at the equipment end, the computer must respond to a request for service before the data is lost. A multiprogramming system with terminals operating under strict time constraints is called a real time system. Often real time systems have hundreds of terminals, each of which requires service within microseconds of the time a request is posted.

Yet another species is the network or distributed system. These handle substantial numbers of CPU's (> 10). The distinction between a network and a distributed computer is one of computational intimacy. If the CPU's are physically far apart, and working on different problems, it is regarded as a network (e.g. FARBER's ring network [1]); if they are physically close together, working on the same problem it is regarded as a distributed system (e.g. the CMU Multimini, WULF [2]). These types of systems are new, but rapidly increasing in importance.

Lastly there are dedicated systems, which do not attempt to support general programming, but are specific to one application. Data base management, banking, and computer aided instruction are typical application areas.

1.2. *Characteristics of multiprogrammed operating systems*

First, they are huge. OS/360, for the IBM 360's and 370's consists of 30,000 pages of assembly code. It took 5000 man-years to construct it. The MULTICS system, for the Honeywell 6180 consists of 4000 pages of PL/I. Their friends and relatives are equally large.

Second, they involve parallelism, both in hardware and software. Many simultaneous activities must be coordinated.

Third, they are nondeterminate. If two travel agents simultaneously try to reserve the last seat on an airplane, one will win and one will lose (hopefully). Operating system behavior is not reproducible, unless timing considerations down to the nanosecond level are taken into account, and even then, the problem of arbitration of exactly simultaneous events must be considered.

Fourth, multiprogramming systems must provide facilities for permanent long term storage of information (file system). If the owner of a 100,000 volume medical library available from remote terminals had to retype the entire library every day, he would probably not be wildly enthusiastic about his operating system.

Fifth, multiprogramming systems must provide for the controlled sharing of information and resources. There is a crucial distinction between technological sharing and intrinsic sharing. Technological sharing is done for economic reasons only. When CPU's become cheap enough to give every program its own (1980-1985), there will be no need to share CPU's among programs. On the other hand, the sharing of the passenger lists in an airline reservation system among the various programs (one per travel agent) is essential.

No improvement in technology will eliminate this sharing (except maybe giving each passenger his own airplane).

In a large system, with many independent user groups, adequate facilities for, and controls on, sharing is of great importance. The MULTICS project began as a prototype of a national or regional computer utility, which would provide 24-hour a day continuous service on a metered basis, much like the telephone company or electricity company. This would eliminate the need for potential computer users to start out by buying or renting an entire computer, with its attendent space, staff and maintenance headaches. In a computer utility, providing adequate facilities for, and controls on, sharing is probably the dominant design consideration.

Sixth, multiprogramming systems must automatically manage the allocation and use of the various hardware facilities, without expecting much help on the part of the users

1.3. *Services provided by typical operating system*

One view of an operating system is that it provides its user with a virtual machine, one that is less awkward that the bare hardware. This is accomplished by implementing a class of instructions and features not present in most third generation hardware. Special instructions are usually invoked via a "supervisor call" or "emulator trap" instruction.

One major feature provided by most large scale OS's is virtual memory. Programs can address a large address space - often larger than the machine's physical memory - with the mapping of virtual to real addresses handled by hardware, except when the addressed piece of program is not in primary memory. In this case the OS intervenes to fetch it.

Virtual instructions to read and write files are common. Files may be read or written sequentially or randomly. OS/360 provides a multitude of access methods and options, including automatic buffering, deblocking, queueing, and searching. An OS must also provide for creating, destroying, protecting and cataloguing files, as well as many other functions relating to file directory management. Users have come to expect all file operations to be device independent, so that a program written to accept card input will also run using tape input.

All operating systems provide a Job Control Language (JCL) which allows the user to communicate with the OS. Typical JCL's look like assembly language, with one instruction and its parameters per line. Instructions are

things like call a compiler, execute a program, print a file, request a resource (primary or secondary memory space, tape drive, plotter, etc.), produce a memory printout, catalog a file etc. It is an unfortunate property of most OS's that the JCL interpreter is buried deep inside the OS. A better strategy would be to provide "virtual instructions" for all the available facilities, thus allowing anyone who wanted to, to provide his own JCL compiler or interpreter.

2. PROCESSES

2.1. *Characterization of a process*

The concept of a process is central to understanding how an OS works. A *process* is a program in execution. At every point in time a process has a certain *state*, consisting of the program, the values of all its variables, registers, program counter etc. As the process runs, its state changes; the variables take on new values, the program counter advances, and so forth. It is useful to distinguish that part of the process that does not change in time (e.g. the program itself, assuming it is reentrant, and perhaps certain tables) from the part that does change in time. The changeable part is called the *state vector*.

One way of looking at a process is to say that a process consists of 2 parts: The program (including fixed tables) and the state vector. The program is a set of rules describing how the state vector is to be changed in time. The CPU is regarded as an engine that forces the state vector through the sequence described by the program. Another viewpoint regards a process as a (past and future) history of a program's state vector. Either way, a process is an active entity. It can cause events in the outside world to occur, for example, writing chinese on the plotter, or playing chess with another process or being. In contrast, a program is a passive entity. It just sits there and does nothing.

Note carefully that a process is a strictly sequential entity. There is no parallelism within a process.

In order to keep track of the status of each process under its control, an OS must maintain a small table for each process. This table, sometimes called a Process Control Block (PCB), contains information such as the the process' status (running, ready to run, blocked), primary memory allocation

(the memory contains the program and variables), files in use, i/o device status, accounting information (i.e. process history), program status word, quotas (e.g. maximum disk space allowed) etc. Whenever a process is suspended for any reason, its PCB must be updated, in order that it can be restricted in the same state that it stopped in. In fact, the contents of the PCB's in a particular system is an operational definition of what processes consists of. It should be clear now that a program is but one of many components of a process. In fact, from the OS's point of view, the function carried out by the program is irrelevant, only its resource usage and demand for services is visible.

Although sharing of reentrant code is sometimes possible, each process normally runs in its own private address space. No process can just reach into another process' address space and examine or change data, except by prior consent. Protection mechanisms use this feature heavily.

2.2. *Interprocess communication*

Processes need to communicate with other processes for a variety of reasons. One of the most important is to communicate information. For example, in an OS consisting of several processes, there is likely to be a process that manages the line printer. All printing is handled by the line printer process. To have a file printed, a user process must somehow convey to the printer process the fact that it wants a file printed, and which one. Situations like this are very common.

Another reason interprocess communication (IPC) is needed is to allow processes to synchronize their activities. As an example consider a producer-consumer problem with a shared buffer capable of holding only a single item. The sequence of execution must be: producer fills buffer; consumer empties buffer; producer fills buffer; consumer empties buffer, etc. In other words, the two processes must communicate with each other in order to strictly alternate their buffer accesses.

A somewhat related problem is that of mutual exclusion. In many systems certain data bases must only be accessed by one process at a time, although the order of access is not important. A typical example is a flight list in an airline reservation system; bad things could happen if two processes tried to reserve one seat simultaneously. From the viewpoint of the processes and the OS, it does not matter which of several interested processes gets the data base first, so long as no more than one has it at any given time.

All interprocess communication involves sharing of some address space (or object upon which an instruction operates) between the communicating processes. Sometimes this is disguised, as in the case where all processes can (indirectly) read and write from a common table within the OS. Three of the many interprocess communication mechanisms in use are described below.

2.2.1. *Semaphores*

Excluding very low level communication primitives, like test-and-set-lock-or-skip type instructions, DIJKSTRA's semaphore system [3] is about as simple as IPC can be. To communicate, processes arrange for the existence of a protected, non-negative integer variable (semaphore) shared between (among) themselves. Two instructions on the semaphore are provided by the OS: *down* and *up*. If a process does a *down* on a positive semaphore, its value is simply decremented by 1. If a process does a *down* on a zero valued semaphore, it is blocked and the *down* is not completed. If a process does an *up* on a semaphore , its value is increased by 1. However, if another process was blocked on the semaphore, it can now complete its *down* instruction and proceed. If several processes were blocked on the same semaphore, one of them is chosen by magic to continue. The others remain blocked.

Semaphores are simple to implement. The OS need only maintain for each semaphore a list of processes blocked on it. If a field is provided in the PCB, a semaphore can be represented by two items of information: a value, and a pointer to the first PCB linked on the blocked list for that semaphore. Subsequent processes blocking on the same semaphore are linked together via the PCB field.

Semaphores also have a severe disadvantage: processes can use them to synchronize control, but not to pass information. The example of a process wanting to print a file given above would be impossible to implement using semaphores alone; a second IPC mechanism (e.g. address space sharing) would be needed in addition.

2.2.2. *Messages*

With this IPC mechanism, every process has a mailbox associated with it. Two primitive instructions are provided: *get* and *send*. When a process executes a *send* instruction it specifies a destination process and message buffer within its own address space. The OS then copies the message to the

receiver's mailbox, and the sending process is then free to continue. The mailboxes are maintained within the OS itself, so no process can examine its own or any other process' mailbox. To get a message, a process executes a *get* instruction. If there is a message waiting, the OS it copies into the receiver's address space, along with the sender's identity, provided by the OS, so as to be unforgeable. If no message is waiting, the receiver blocks until such time as a message arrives.

One can regard each mailbox as a private semaphore associated with each process. Doing a *get* is analogous to doing a *down*; sending a message to a process is analogous to doing an *up* on its semaphore. The number of messages in the mailbox is analogous to the semaphore value. Many variations of this popular scheme exist, such as the ability for two processes to create a private tube (event channel) between them for sending and getting messages.

Messages have the obvious advantage of combining communication of information with synchronization. They have the disadvantage of requiring a space consuming message table within the OS. If the table threatens to fill up, processes attempting to send messages can be retroactively suspended (at the *send* instruction) until there is more room.

2.2.3. *Monitors*

A *monitor* (BRINCH HANSEN [4]) is a data structure consisting of local data, and 1 or more procedures that processes can execute. Monitors have the property that only 1 process at a time may be invoking a given procedure. There is a special data type called a condition. A process may execute a *wait* instruction on a condition variable, which causes it to block until another process executes a *signal* instruction on the same variable. For example, here is a monitor to allocate and release a single dedicated resource.

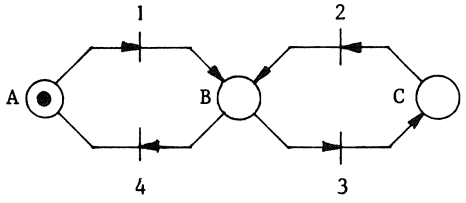
```
monitor resource =
begin bool busy := false; condition flag;
      proc acquire = void: (if busy then wait flag fi; busy := true);
      proc release = void: (busy := false; signal flag)
end
```

Monitors may contain procedures and initialized local data. To acquire the resource, a process executes acquire. If the resource is in use, the flag busy will be true and the process will execute the *wait* flag instruction,

blocking itself. When some other process calls release, it will *signal* flag, freeing the blocked process to continue.

2.3. *Petri nets*

Multiprocess systems can be modeled by Petri Nets. A *Petri Net* (PETRI [5]) is a directed graph containing two kinds of nodes, places (circles) and transitions (bars). There are a finite number of tokens distributed among the places. The Petri Net moves from state to state in time. A transition is said to be enabled if every input place has at least one token. At every step in time, exactly 1 transition fires, removing exactly 1 token from each input place, and depositing exactly 1 token in each output place. Tokens are not conserved. The following Petri Net models mutual exclusion.



A token at A indicates process 1 is running; a token at C indicates that process 2 is running. A token at B indicates that the OS is about to choose a process to run, by the nondeterministic firing of either transition 3 or 4 but not both. The firing sequence 141414141414 shows process 1 hogging the CPU, whereas 132413241324 is a round robin scheduler.

3. OPERATING SYSTEM STRUCTURE

3.1. *The big mess*

This is the traditional form used by nearly all commercial systems. It consists of having thousands of procedures randomly calling one another. The result is systems that crash 3 times a day, leaving the systems programmers to wonder how they work at all.

3.2. *Hierarchial systems*

This system design was made popular by DIJKSTRA's THE system [6]. It is essentially a bottom-up design in which a virtual machine is built up in layers, each layer adding new features to the virtual machine used to con-

struct the next higher layer.

At the bottom level is the bare machine, which is nondeterministic due to interrupts. The flow of control is unpredictable because any instruction may be followed by the first instruction of some interrupt routine. The next level has processes that communicate via semaphores. Each process is strictly sequential and deterministic. A hardware interrupt merely causes the CPU to switch from whichever process it was running to the interrupt process. Subsequent layers add features such as virtual memory, virtual terminals, virtuals i/o instructions etc.

3.3. *Extendable systems*

Another design technique, made popular by the RC4000 system [7] of BRINCH HANSEN is that of providing a common nucleus for different OS's. In the case of the RC4000 OS, the common nucleus is based on a variation of the message handling IPC mechanism. The nucleus provides certain basic facilities to higher level processes, which can be used to build OS's.

As an example of an extendable system that is also hierarchically structured, we note a time sharing system for the PDP-11/45, TSS-11 (TANENBAUM [8]). In this system, the facilities available are gradually increased over a number of levels until the "genie" level is reached. A process running at the genie level has virtual memory, IPC via messages, virtual i/o (by sending messages to the terminal handling processes, by sending file descriptors to printer, punch, etc. processes) etc.

Associated with each user process is a genie process, the latter of which runs at the genie level. The only operation a user process is capable of executing is sending a message to its genie. The genie then carries the request out by sending messages to lower level processes. Different users may simultaneously have different genies. Since the nature of the user machine's virtual instructions, file system, JCL, etc. are determined by the genie, different users may appear to be running on different and incompatible OS's simultaneously.

3.4. *Self virtualizing machines*

A novel and extremely powerful technique for structuring an OS follows from two observations: 1) In a multiprogramming system, each user has a subset at the "real hardware" instructions, like register add, and a set of "virtual" instructions, like read file. The former are carried out by the

hardware or microprogram, and the latter are carried out by the OS; 2) If the OS is working properly nothing any user process does can damage any other process.

In a self virtualizing machine system (GOLDBERG [9]), the virtual machine presented to the user is an exact duplicate of the complete real machine, including all the i/o instructions, ability to change or circumvent the protection mechanisms etc. Each virtual machine runs its own complete OS. This is accomplished as follows. To use this technique one needs a machine with two modes - user mode and system mode, in which all sensitive instructions (i/o, fiddling with the protection, changing the memory, mapping etc) are trapped in user mode. In a standard OS, when a user process executes a sensitive instruction, a hardware trap to the OS occurs, and the process is abruptly aborted with a suitable rude message such as INV INSTR AT 43B7A.

In a self virtualizing machine system the OS running on the real hardware forces all virtual machines to always run in user mode, but it keeps careful track of whether each virtual machine is in "virtual user mode" or "virtual system mode". When a virtual machine executes a sensitive instruction in virtual user mode, the true OS running on the real machine (called a virtual machine monitor, or VMM) gets control. Instead of aborting the offending program, it passes control to the OS in the virtual machine causing the trap. This OS then does whatever it normally does when a user process does something wrong, such as abruptly aborting the process.

However when a virtual machine in virtual system mode executes a sensitive instruction, the VMM handles the trap completely differently. It simulates the instruction. If it was an i/o instruction, the VMM carries out the i/o (using virtual card readers, virtual disks, etc.) bit for bit the way the real hardware does. When the sensitive instruction has been completed, the virtual machine can continue.

Note that there is no way for a virtual machine to damage its neigbours. It can erase the non-resident portion of its own operating system and destroy all the files on its virtual disk, but it cannot affect any other virtual machine's virtual disk.

This design has effected a complete partitioning between the multiprogramming and user command interpretation functions of an OS. The VMM handles all the multiprogramming, and the simulation of the sensitive instructions (of which there are usually fewer than 10). The OS's in the virtual machines can be simply monoprogrammed systems that only need worry about interpreting

and executing user service requests. By splitting an OS into two very distinct parts, it becomes much more manageable. This type of system is available on the IBM 370's under the name VM/370. Certain defects in the 370's design made writing the VMM more difficult than it would otherwise have been (the 360's designers back in 1962 never envisioned such a system), but it is easy to design a machine to make VMM's easy to write. Future computers will probably use this technique heavily.

3.5. *No operating system*

In MULTICS, there is no distinction made between the system processes and the user processes. Each user process has as part of its address space (albeit protected) routines to schedule the CPU, handle page faults, allocate disk space, etc. Thus each user process is self supporting, never needing to invoke "the system" for anything. Of course this does not address the issue of how the protected routines within each process are structured (hierarchically, as it turns out), but the idea is intriguing.

4. RESOURCE MANAGEMENT

4.1. *CPU management*

One resource that all OS's must manage is the CPU. Usually there are several processes competing for the CPU. The OS must have some policy about deciding in what order to run the processes. In a multiprogramming system with no time sharing users a common scheduling algorithm is Run To Completion. Each job is given an external priority (e.g. professors get better service than students, rich people get better service than poor people, everybody gets so many minutes of "big rush" time to use when he needs it etc.). Whenever the CPU is idle, the highest priority program is given the processor. It continues running until it finishes or blocks for i/o. Then the next highest priority program is run. A variation of this scheduling algorithm has the priorities dynamically adjusted by the OS. For example, highly i/o bound programs could be given high CPU priority to make sure their i/o keeps going continuously.

Another common algorithm is Round Robin. All the programs bidding for service are given short quanta (e.g. 100ms) in succession. Four programs would run in the sequence 123412341234. In Run To Completion, a greedy pro-

cess can get the CPU and keep it for 2 hours; Round Robin prevents this.

Another well known scheduling algorithm is the Feedback Queue system. The scheduler maintains N queues, the highest getting the best service. When a process must be chosen, the first process on the highest populated queue is selected. If a process terminates by blocking (for input) it is removed from the queue system. When it later unblocks, it is re-injected on the highest queue. The result of this algorithm is to keep the highly interactive processes on the top queue, and the more compute bound processes on successively lower queues.

Yet another scheduling algorithm attempts to give process i N_i seconds of CPU time per minute, no more and no less. Such an algorithm tends to give the user uniform response, independent of system loading.

Butler LAMPSON has suggested that next to each terminal there be a large red button. If a user is unhappy with the service he is getting, he may opt for pushing aforesaid button, in which case one of 2 things happens: either his service gets better or he is automatically logged off. The CPU time acquired by logging people off in this fashion goes into a reserve pool to be used for providing better service to the winners of the big red button show. The idea is that most people are reluctant to leave their terminal, even when the service is bad and they have other work to do.

4.2. *Memory management*

OS/360 and some of its imitators divide primary memory into a number of partitions. In some versions the number and size of the partitions is keyed in by the console operator in the morning and remains fixed all day. In this type of system each program is loaded into some partition where it remains until finished. In other versions the number and size of the partitions is dynamically variable. Sometimes the OS can compact memory by squeezing the partitions together, in order to gather all interpartition space into a single hole, hopefully large enough for another program. Moving programs around can be tricky, however, if i/o is in progress. Compacting outward from the middle is more efficient than "top down" or "bottom up".

For time sharing systems, programs must frequently be swapped out to secondary memory. Since it is usually difficult to insure that they will be put back later at the same address, some mechanism is needed to allow programs to be reloaded anywhere. The most common method is paging. The address space is broken up into units called pages (typically 256-1024 words).

Special hardware defines the memory locations (page frame) corresponding to each page of address space. When a program references its address space, the hardware correctly maps the reference onto the proper physical memory address, or traps to the OS if the page is in secondary memory (page faults).

It is also possible to implement a 2-dimensional address space, by having each instruction specify an address space (segment) and a location within it. This is called segmentation. Paging and segmentation are convenient for the user, giving him the illusion of a large address space, and other advantages. For the OS, paging and segmentation mean more work, keeping track of which pages are where, and managing all the paging traffic between primary memory and (perhaps a hierarchy of) secondary memory.

After a process has been running for a while, the pages being heavily used will naturally gravitate towards primary memory, whereas the less heavily used pages will be kept in secondary memory. The heavily used pages are known as the working set. An OS can attempt to determine the working set of all its processes. When it comes time to run a process, the OS can preload the working set. Alternatively the OS can just start the process anyway, and let it cause repeated page faults, bringing in the pages as needed. This is called demand paging.

When a page fault occurs, the OS must choose a page to remove. This is called a page replacement algorithm. Some algorithms use a load strategy, removing only the faulting process' own pages, while others use a global strategy, removing any convenient page. Popular algorithms include removing the least recently used page, regardless of its usage. A variation removes all the pages of some unlucky process, the argument being that it can't run without its complete working set, so why keep any of it around.

4.3. *Management of rotating memories*

Just as there are innumerable algorithms for managing primary memory, there are innumerable algorithms for page placement in secondary memory, and traffic control between the two. Some systems try to minimize page traffic to keep overhead low; others try to maximize it, to produce a short turn around by keeping processes flowing through primary memory at a high rate.

The simplest scheduling strategy for rotating memories is first come first served. It is also the worst. If many requests are pending when a decision has to be made, one algorithm is to choose the that can be completed fastest. On a moving head disk this may keep the arm shuttling back and forth near the middle of the disk, giving terrible service to pages at the

extremes. The elevator algorithm sweeps from one cylinder to another, changing direction only at the ends. Algorithms taking rotational position into account as well exist for both disks and drums.

Another parameter to be considered is whether pages should always be rewritten in the same place. In some systems a page is rewritten in the first empty position that comes by. This drastically reduces waiting for completion and arm movement, and drastically increases the bookkeeping required to find the page again later.

Since pages that have not been modified since being brought into primary memory (clean pages) neet not be rewritten (a good copy already exists out there), there is a high premium to maintaining a high ration of clean pages/dirty pages. Some systems use idly disk or drum time to rewrite dirty pages, even if there is no specific reason, just to maintain this ratio high. These are called sneaky writes, and whether the overhead required to perform them is worth the gain is an open question.

Note that the previous 3 sections all deal with technological sharing, which can be expected to decrease in importance in the future.

4.4. *Deadlocks*

The CPU and primary memory are pre-emptable resources. If somebody more important comes along, they can simply be snatched away from their current owner. Tape drives, plotters, and certain other resources cannot be effectively pre-empted without abandoning all work currently in progress. Non-pre-emptable resources can lead to trouble. Consider a system with 1 tape drive and 1 plotter. Process 1 asks for the tape drive and gets it. Process 2 asks for the plotter and gets it. Now process 1 asks for the plotter, and is blocked until it becomes available. Unfortunately, process 2 does not release the plotter, instead it asks for the tape drive. Both processes are stuck and will remain so forever. This is called a deadlock.

Three deadlock management strategies are in use: 1) prevent deadlocks by system design, 2) avoid deadlocks at resource allocation time, 3) do not prevent deadlocks; let them occur and have the operator rerun the jobs. If studies show that a deadlock can be expected once every 19 years in a particular system, clearly alternative 3 is preferable to permanently tying up 10k of precious primary memory with somebody's deadlock algorithm, no matter how brilliant.

The most common ways of preventing deadlocks by system design are to

reserve all necessary resources when a process starts, or to require a process to release all previously acquired resources before making additional requests. Strategy 2 above is excellent for generating Ph.D. Theses. All methods require the OS to know the maximum requirements of each process in advance. One algorithm grants or denies resource requests according to this criterion: after each resource grant, there must be some execution sequence that allows each process to complete.

5. PROTECTION

5.1. *Why bother*

If a system had no protection one user could invade the privacy of another. The public would not stand for a system in which salary information, medical histories, police records, etc. were essentially open to the public.

Another problem is industrial espionage. Companies have secret information, both technical and financial, which they do not want their competitors to know.

If a system had no protection, one user could destroy the irreplaceable data of another, or in malicious cases, secretly change it.

Several years ago a California university student found a way to penetrate the telephone company's OS. He formed his own corporation and rented a small computer. Using this computer he called up the telephone company's computer and ordered expensive electronic equipment to be delivered to warehouses around the state at 3 a.m. Shortly after each delivery, he would show up with a truck and steal the equipment. He managed to steal several million dollars worth of equipment before he was accidentally caught. After getting out of jail he became a consultant on computer security.

Not too long ago another computer thief managed to steal 200 railroad cars full of merchandise by convincing the railroads's computer that the cars had been scrapped. It is estimated that computer theft in the U.S. alone amounts to $2\text{-}3 \times 10^9$ dollars/year (ALLEN [12]).

Still another reason why protection is important in OS's (as if the above were insufficient) is that without it one user could degrade the service given to other users, for example by hogging all the disk space.

The problem of protection is greatly compounded by the need for sharing, both technological and intrinsic. When one conceives of a computer utility with thousands of users, some of whom are offering services (programs, data,

etc.) and some of whom are using these services, the problem of allowing only permitted accesses becomes very difficult.

5.2. *Defects in present operating systems*

The protection mechanisms used in most commercial systems are very inhomogeneous. One mechanism is used to protect this, and another to protect that. For example, memory is protected by address mapping, relocation and bound registers, or lock and key schemes. Files and logging in are protected by passwords. Access to peripherals and terminals is protected by internal OS tables. Privileged instructions are protected by a user mode/system mode bit in the PSW, which is in turn protected by yet another mechanism.

It would be much more secure if a single mechanism could be used for all protection. The PDP-11 for example, has no i/o instructions. Instead i/o is performed by setting bits in special memory locations. Thus the memory protection mechanism also protects i/o instructions. A generalization of this idea to the point where memory protection protected everything would be desirable.

Here are some general techniques for compromising security in present OS's. Systems are always in a state of flux. New patches, changes, versions, and releases occur monthly. As a result, a systems programmer could be bribed into putting a "trapdoor" in the system, so under certain very, very rare circumstances, security could be breached.

Many OS's allow processes to request memory, but do not erase it before giving it to the process. By sitting in a loop requesting, examining and releasing memory, a process might discover all kinds of interesting things in the residue, such as the system password table.

Some multiprogramming systems have stringent controls on interactive users, but none on batch jobs. This leads to the Trojan horse attack. Many system search the user's file directory before searching the system directory. A potential spy could submit a batch job to catalog a special version of the editor or loader in the victim's directory. Whenever the victim called the editor or loader, he would get the spy's version, which as a side effect would spy on him and record the information somewhere.

Hitting the user abort key on a terminal often leaves the system in a peculiar state. This is a good target for potential attack.

Files are usually protected by passwords. For users wit a few files, this is adequate, but for a large project with many employees and many files

it is not secure. With 15% annual personnel turnover, a 50 person project will have employees coming and going monthly. A disgruntled employee with knowledge of all the file passwords could wreak havoc. Changing hundreds of passwords every time an employee departed would require a constant stream of memos to all remaining programmers to tell everyone the new passwords. In addition to being an enormous nuisance to the programmers and administrative headache, having hundreds of paper copies of the password lists floating around is obviously a serious security problem itself.

Similar problems exist for a software house that makes a program or data base available, and changes by usage. Preventing customers or competitors from copying the program is nearly impossible with present OS's. Furthermore, taking back permission from customers who terminate the service but continue using the computer for other reasons is well nigh impossible with password protection.

5.3. *The confinement problem*

As a test to see how much protection an OS offers, consider the following problem (LAMPSON [11]). There are 2 processes involved, the service and the customer. The customer makes contact with the service and provides it with information. In return the service performs some computation, and gives the customer the result. The cost of the service rendered is given to the service's owner, who then mails a bill to the customer's owner. (Owners are human beings, not processes.) One such service is a program to help people with their income tax. Neither process trusts the other. The customer might steal the tax program, and the tax program might steal the customer's financial information.

Some of the ways the service might leak information to a third party are quite subtle. If the service has memory, it can store the information for later collection. If the service can create permanent files, it can store the information there. If the service can create a temporary file and grant access permission to a third party, the third party can read the file quickly while the service is still active. The service might be able to leak via the interprocess communication facility. The service could encode information in the bill sent to its owner. The service could lock and unlock a file in a clocked manner, so a third party process could acquire information by continually checking to see if the file were locked, because the time function locked(t) is a binary bit stream. Worst of all, the service could

degrade system performance (by heavy paging, CPU usage etc.) in a clocked manner. A third party could notice the degradation and decode it as a binary bit stream. Of course this channel is very noisy and has a low bandwidth, but information theory techniques could be used to insure reliable transmission.

5.4. *Protection mechanisms*

In MULTICS and some other advanced OS's, a user can map a file onto a segment, i.e. the file becomes part of the virtual memory. This allows the segment protection mechanism to be used for active files as well. One protection mechanism has a protection matrix encompassing all active segments. The entry M_{ij} specifies the access segment i has to segment j (read, write, call, etc.). Unfortunately, the matrix M is too large to be of much use in most systems.

In some systems protection on segments is provided within each process, with the possibility of shared segments. In MULTICS each segment has a ring number, the lower the ring number the more protected. If the current procedure is in ring i, any attempt to access ring j where $j < i$ will be trapped by the hardware. The OS itself operates in each process' address space in ring 0. A user wishing to build a debugging system to debug his own programs can put the debugger in ring k, and the debuggee in ring $k + 1$, thus protecting the debugger.

A number of recent systems use the concept of a domain as a protection environment, a generalization of the MULTICS rings. In such systems, there exist various classes of objects - processes, files, segments, mailboxes, i/o channels, terminals, peripherals etc. Withing each domain, a certain set of objects is accessible, and with certain strings attached e.g. a read only file. Each process is in some domain at each moment, and has access to all the objects of that domain. Note that the same object may appear in several domains with different permissions.

One can imagine a giant matrix whose rows are the domains and whose columns are the objects. Each entry tells what access the domain has to the object, if any. There are 2 practical ways to implement this scheme. First, associated with each domain is a list of all the objects it has access to. This list contains all the nonzero entries in its row of the matrix. Each item in this list is called a capability (DENNIS & VAN HORN [12]).

A capability has 4 parts: a unique number (e.g. date and time the ob-

ject was created, expressed in microseconds elapsed since 0000 GMT, 1 Jan. 1901), the object type, the access bits, and a pointer to the object (its PCB, disk location etc.). The object itself also contains the unique number. If an object is destruyed and the space it occupied reused, an attempt to access it via an old capability will be detected because the unique numbers will not match.

The access bits depend on the object type. For a file, *read*, *write*, *execute*, *copy*, *give away*, *destroy* and *extend* might be reasonable. For a process, *stop*, *destroy*, and *communicate with* might be appropriate accesses. In some systems users or at least subsystem writers, can create new object types, and define what the access bits mean.

Needless to say, the protection system would be worthless if a process could manipulate its own capability list. The capability lists must be maintained by the OS. Whenever the OS creates a new object for a process (e.g. a file) it inserts the capability for the new object in the process' capability list, and returns the index, i, of the new capability (i.e. its position in the C-list). To refer to an object, a process uses the index of the capability. Using this scheme, all objects are protected in a simple, homogeneous way. Capabilities can also be used for accounting by using some of the access bits for integer values. For example, to use the CPU one would need a capability for it, one of whose fields was the CPU time allowed.

The other way to slice the domain-object matrix is by columns. Associated with each object is a procedure which is invoked on every access. The procedure could use any method it wanted to in order to restrict access (such as asking permission from a logged in user).

5.5. *Fundamental principles of protection*

Protection should be based on explicit permission, not exclusion. The default is no access. Among other things, security failures generally show up in the form of prohibiting an allowed access instead of allowing a forbidden one. Check every access for current authority; otherwise permission gets stored away in local memories. The design should be public. Protection should not be based on the assumption that a potential attacker is ignorant. Do not give anyone more access than he needs. The human interface must be easy to use, or people will not use it. Some users want strange things; provide him with facilities to create his own protection subsystems.

5.6. *User authentication*

In MULTICS, each registered user of the system has his own private password. That is the only place passwords are used. The system keeps track of who may access what (e.g. associated with every file is an explicit list of users allowed to access it). Once you are logged in as JANSEN you can do everything JANSEN can do.

To reduce the chance of one person logging in as another the following measures are taken. Terminals do not echo when passwords are typed, so there is no written record of the password. A user may change his password at any time. Not even the system administrator or computer center director can discover a user's password. Passwords are stored internally in an encrypted form that cannot be inverted. This list could be posted on the terminal room wall and it would not compromise system security (see EVANS, KANTROWITZ & WEISS [13]). To discourage users from picking easy to guess passwords, the system supplies a random password generator that uses English diagram frequencies (e.g. foat, zabel, norbid but not qfupz, hqiznp, jjaqrp). All batch jobs must be started by a logged in user. After a period of N minutes of inactivity and after all crashes, the user must log in again. The system forcibly breaks the telephone connection after 10 unsuccessful log in attempts (to make random searches harder). All login's are recorded, and the time and place of the previous one is printed after each login (if someone else logged in as you, you will at least be aware). Lastly, a user can be set up to run a specific program after login. This program could begin asking questions, etc. to give a user even more protection against an intruder.

Other authentication systems used elsewhere include the following. Passwords that are good for one use only, with the new one being typed at logout, or presupplied by user. (Stealing used passwords has no value). Special terminals that require insertion of a magnetically striped card during password type in. Systems that hang up after login, and automatically call the user back (this requires the intruder to get into the user's office).

6. PERFORMANCE MONITORING AND EVALUATION

Operating systems are so complicated that it is difficult to tell how well they are working. In order to tell if a new version is better than the old one, it is necessary to make extensive measurements on both. Such mea-

surements include things like CPU idle time, time spent in various processes, including system activities (IPC, scheduling, paging etc.), virtual/real memory ratio, paging behavior etc.

6.1. *Measurement tools*

Some of the basic tools used for performance monitoring are as follows. Counting invocations and time spent in each OS procedure. Histograms of the program counter sampled every T milliseconds (clock interrupt). Records of page and segment faults. Event tracing (messages sent, interrupts, i/o, process switching etc.). Use of an external computer to display statistics in real time. Program sitting in a tight loop reading the clock, and logging all gaps (this can measure interrupt handling time, and find scheduler errors). The ability to have charged CPU time, number of page faults, etc. printed on each terminal after each command. To reproduce input behavior for making tests, an external minicomputer attached to several telephone lines is best.

When constructing a synthetic workload, one method is to use a known mix of assemblies, compilations, sorts, matrix inversions, etc. Another method is to characterize each job by its demand of system resources, e.g. CPU, memory, page faults, file usage, etc. Thus each job can be characterized by an n-tuple. Run the OS for a while, and collect statistics on these n-tuples. Use these to construct the probability density function in n-space. Write a test program that demands services randomly according to this p.d.f.

6.2. *Modeling*

Analytic models of OS behavior are useful for understanding complex systems. For example, should a computer center buy a 500 nsec CPU with 500k primary memory or a 200 nsec CPU with 300k primary memory for the same price?

To give an idea of how modeling can be used, consider a system in which the mean time between page faults is linearly proportional to the primary memory, M, allocated to a process' pages, i.e. 1 page fault every aM sec. Assume a normal instruction takes T_1 and a page fault causing instruction takes T_2 sec. The average instruction time, $T = T_1 + T_2/aM$ sec/instr. Suppose the machine rental cost is $C_1 + C_2M$ guilders/sec, where C_2 accounts for memory charge, and C_2 everything else. The cost per instruction is then $(C_1+C_2M)(T_1+T_2/aM)$. Minimizing this with respect to M we find that optimal memory size is $(C_1T_2/aC_2)^{\frac{1}{2}}$.

REFERENCES

[1] FARBER, D.J., *A Distributed Computer System*, Report TR-4, Dept. of ICS, Univ. of Calif., Irvine, 1970.

[2] WULF, W., E. COHEN, W. CORWIN, A. JONES, R. LEVIN, C. PIERSON & F. POLLACK, *HYDRA: The Kernel of a Multiprocessor Operating System*, CACM 17 (1974) 337-345.

[3] DIJKSTRA, E.W., *Cooperating Sequential Processes*, in Programming Languages, F. Genuys (ed.), Academic Press, New York, 1968.

[4] BRINCH HANSEN, P., *Operating System Principles*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

[5] PETRI, C.A., *Communication With Automata*, Rome Air Develop, Cent., Suppl. I to Tech. Rep. No. RADClTR-65-377, Reconnaisance-Intelligence Data Handling Branch, Rome Air Develop. Center, Griffin AFB, New York, 1966.

[6] DIJKSTRA, E.W., *The structure of the 'THE' Multiprogramming System*, CACM 11 (1968) 341-346.

[7] BRINCH HANSEN, P., *The Nucleus of a Multiprogramming System*, CACM 13 (1970) 238-241.

[8] TANENBAUM, A.S., *A General Purpose Timesharing System for the PDP-11/45*. Wiskundig Seminarium Report IR-2, Vrije Universiteit, Amsterdam, 1973.

[9] GOLDBERG, R.P., *Survey of Virtual Machine Research*, Computer 7 (1974) June, 35-45.

[10] ALLEN, B., *Embezzler's Guide to the Computer*, Harvard Business Review, July-August, 1975, 79-89.

[11] LAMPSON, B.W., *A Note on the Confinement Problem*, CACM 16 (1973) 613-615.

[12] DENNIS, J. & E. VAN HORN, *Programming Semantics for Multiprogrammed Computations*, CACM 9 (1966) 143-155.

[13] EVANS, A., JR., W. KANTROWITZ & E. WEISS, *A User Authentication Scheme Not Requiring Secrecy in the Computer*, CACM 17 (1974) 442-445.

READING LIST (BOOKS ON OPERATING SYSTEMS)

BRINCH HANSEN, P., *Operating Systems Principles*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

BROOKS, F., *The Mythical Man Month*. Addison-Wesley, Reading, Mass., 1975.

COFFMAN, E.G., JR. & P.J. DENNING, *Operating Systems Theory*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

FREEMAN, P., *Software Systems Principles*. Science Research Associates, Chicago, 1975.

KATZAN, H., JR., *Operating Systems: A Pragmatic Approach*. Van Nostrand Reinhold, N.Y., 1973.

MADNICK, S.E. & J.J. DONOVAN, *Operating Systems*. McGraw-Hill, N.Y., 1974.

ORGANICK, E., *The Multics System*. MIT Press, Cambridge, Mass., 1972.

SAYERS, A.P., S. KURZBAN & T.S. HEINES, *Operating Systems Principles*. Petrocelli/charter, N.Y., 1975.

SHAW, A., *The Logical Design of Operating Systems*. Prentice-Hall, Englewood Cliffs, N.J., 1974.

TSICHRITZIS, D.C. & P.A. BERNSTEIN, *Operating Systems*. Academic Press, N.Y., 1974.

WATSON, R.W., *Timesharing System Design Concepts*. McGraw-Hill, N.Y., 1970.

MATHEMATICAL CENTRE SYLLABUS 32, 1976, II, 25-41

VIRTUALITY

H.J. BOOM

1. VIRTUAL STORAGE

Once upon a time, some people wanted to build a computer with lots of memory, but they had very little money. They realized, however, that they could use disk or drum storage if they had to, because disks and drums were cheaper than main memory. But this raised software problems. If a program used a data base too large to be kept in core in its entirety, it would have to keep it on disk and pay the extra price of disk transfer time and program complexity. It frequently happens that a program design is based on the speed of the disk drive instead of the problem it is to solve. It becomes necessary to build a certain decision very solidly into each program: which data reside on disk, and which reside in main store. Large amounts of data can fit only on disk; frequently accessed data are wanted in main store. If the division of data among disk and main store were badly done, program capacity and efficiency would be seriously impaired, perhaps by several orders of magnitude. Furthermore, the characteristics of data usage might well remain unknown until the program has been written, debugged, used, and measured. At this time, a change in data distribution can require a complete rewrite of the program. Even worse, after the program has been rewritten, the pattern of usage may change, requiring yet more rewrites.

Sometimes programmers look at a problem in despair until they catch a glimpse of a solution.

The individual in the best position to know which data are used most often is the program itself. It need only count. If all accesses to the data base are made through a central module, the module can keep usage statistics for the various data and arrange that the most-used data are maintained in main store. If in any run data usage is sufficiently concentrated in one part of the data base, very few disk requests will be needed, even though each run of the program may have different data usage patterns.

On the other hand, there is significant extra overhead. All data accesses must go through a separate module to be re-interpreted appropriately, usage statistics must be kept, and tables must be maintained as to the position in main memory or disk of each datum. This is considerably more work than simply fetching a word from main store, and once again the programmer is tempted to divide his data into "fast" data which he himself maintains in main store and "slow" data which he leaves to this system. And once again, if he guesses wrong... .

How can one overcome the overhead? Nowadays, the phrase is "let the hardware do it". Because of the cost of hardware, one did not think of this so readily in the old days, but nonetheless, this solution was used.

The design of such hardware had to be based on several principles and constraints.

- Most requests would have to be satisfied from main store. Otherwise, there would be no performance improvement over the software solution, and the hardware cost would not be justified.
- Hardware was very expensive. Only those parts of the access algorithm that had to be fast could be afforded in hardware. For operations that could be done slowly, software could be used.
- Access to data in main store had to be fast; disk access could be slow. The disk itself was slow anyway.
- All storage fetch operations, including instruction fetching, should be done using this mechanism. Then programs could be large as well as data, without any specific provision for program overlays.

The scheme used is as follows:

- The data base is divided into blocks called "pages".
- Each storage reference directly requested by the execution of a program instruction requires the reading from or writing to some specific "virtual" storage location, given its address.
- This address is broken into two parts, the "page address", which identifies the page, and the rest, which identifies a datum within the page.
- The page address is used as an index into a table.
- Each entry in the table contains
    - the address in main storage where the page resides, or
    - the address on disk, and also

  - a bit which indicates which of the above two possibilities holds.
- If the page is in main storage, it is used directly; otherwise, a standard operating-system routine is used to ensure that the page comes into main storage

"Virtual storage" was born.

A few properties should be noted:

- It is not necessary that the size of addresses used in user programs have anything to do with the amount of main storage. There may be more virtual storage, just as much, or less virtual storage as main storage.
- The user is concerned with the "virtual addresses" the system makes available to him, and not the "physical addresses" or "real addresses" of the underlying hardware. The operation of the paging system should be as important to the user as the kind of transistors in the NAND gates. It does make a difference in performance, but it does not otherwise change specifications.
- On modern machines, various table structures are used and various other tricks (such as small fast associative memories) are used to reduce the time lost in looking up page table entries.
- The programmer can now free himself from explicitly coding and optimizing operations on backing store. If he wishes, however, he can still arrange his storage references so as to reduce paging.

2. VIRTUAL CPU'S

Sometimes it is simplest to think of a job as consisting of a number of independent programs running independently on different CPU's. For example, a number of people may wish to communicate with programs via individual terminals. Although it is sometimes possible to have them all communicate with one large centralized program, it is simpler to create "virtual CPU's", one for each user's program.

If one has enough money and cheap enough hardware one can simply buy a number of separate physical CPU's. If not, one is in the same position as with memory. Rather than have a single piece of memory be dedicated to a single datum, even though another datum on disk may be more urgently required,

virtual storage was invented. In a similar way, we invent "virtual CPU's". A CPU may not always be actively being used by a program; sometimes the program may be awaiting input or output. At such times, one may as well run other programs. This trick is called "multiprogramming" if scheduling is also done by the machine (usually by software).

Given a reasonable scheduler, with timer interrupts and other machinery, this transforms one physical CPU into a number of virtual CPU's. Just as with virtual memory, the virtual CPU's need not be quite identical to the real CPU. In particular, the speed of execution, measured in real time, will not be constant.

The number of virtual CPU's need bear no fixed relation to the number of physical CPU's. It may be greater, equal, or smaller and it may vary from time to time. The virtual CPU's need not have all the capabilities of the original CPU either, but we shall speak more of this later.

3. PROTECTION

When one builds such a multiprogramming system, one is faced with a choice. Either one regards the programs to be run as reliable or one does not. In practice, it is usually better to regard them as unreliable. They will be unreliable during debugging; and it is a rare program that is ever frozen permanently at any stage of development.

Let us consider a relatively awkward case. Let the various programs share a common data base, which they must maintain and update. It is no longer possible to consider these programs on the various "virtual CPU's" as independent. We can consider two kinds of failures:

- incorrect modification of the data base, using correct methods of access.
- wild actions which demolish the data base or the other programs (for example, a store into memory using an incorrect address).

Complete system reliability requires that both kinds of failures not occur. However, it is far easier to catch the second kind of failure than the first. It is this second kind of failure that we shall discuss here.

Third generation computers often came with two operating modes, a "system" mode and a "user" mode. In the "system" mode, all machine operations are available. In the user mode, certain sensitive operations are not available. These sensitive operations are usually those involving input/

output, interrupt handling, and virtual storage management. To have one of these operations executed, the user program must first enter the system mode. The only way for it to do so is to invoke an operating system routine. This may be done via a special instruction (a "system call"), or by an interrupt (such as a paging interrupt).

This ensures that the user's program can execute only certain procedures in system state. This simplifies the problem of ensuring reliability. Only the system-mode routines need be certified to prevent users from interfering with each other (except for misuse of normal interfaces, of course).

Prevention of wild stores into memory can be accomplished by controlling the paging map.

4. VIRTUAL INPUT AND OUTPUT

Let us consider examples of virtual resources.

Each has the property that direct access to the real resource is prevented or monitored. For virtual memory, there is a paging map to separate the real CPU's and a user state which prevents direct access to privileged hardware facilities.

The next step is virtual input/output. We have to provide virtual card readers, printers, disks, magnetic tapes, etc. When this has been done, we may be able to run a complete conventional operating system on one of our virtual machines, and another on another simultaneously. The only problem will be one of capacity - will two systems try to take over more than 100% of the machine's capacity?

Virtual input/output requires that virtual input/output operations be remapped onto real operations by hardware, software, or both. For some devices, it may be convenient simply to use a device map analogous to a paging map. However, what is normally done is to trap all input/output operations and then re-interpret them in software. This makes it possible to build virtual machines with input/output devices different from the real machine. This may be useful, for example, when debugging a file management system - its routines can be tried on devices that are not physically present at the installation.

In a sense, ordinary operating systems usually provide such virtual devices, from disks they create virtual devices with catalogs, indexes, random access and indexed sequential files, and so forth. The main differ-

ence between these conventional systems and what one normally thinks of as "virtual" is that the virtual facilities do not greatly resemble the real ones.

5. VIRTUAL MACHINES

Let us generalize. In general, we may say that in a virtual system, access to resources is monitored or restricted in such a way as to enable the substitution of some resources for others. Much as a man looking through a telescope sees a virtual image instead of a real one, a user of a virtual computer system sees the virtual one instead of the real one. The fact that his system is virtual has certain effects on performance and cost, much as the virtual image in the telescope differs from the real one.

The idea of virtuality can be extended to the entire computer system. As long as user programs must be executed in a "user mode" that makes certain operations inaccessible to the user directly by causing them to interrupt to a supervisor instead, it is possible for the supervisor, which operates in "system mode", to substitute other facilities. There exist operating systems which provide virtual CPU's, virtual memory, virtual I/O devices, virtual supervisor and user states, etc.

The user actually remains in user mode. Whenever he attempts to exceed his capabilities of the moment, a trap to the supervisor occurs. The supervisor can then decide whether to carry out the operation, suppress it, abort the user job, or do something else.

The net result is that in such a virtual system, the user can act as if he has a bare machine to himself. Except for timing, the existence of the virtual machine operating system should be quite transparent.

6. VIRTUAL TIME

Such a virtual system should have two clocks, a time-of-day clock and a CPU clock. The time-of-day clock must always continue running; the CPU clock would stop ocassionally, such as when the operator presses the STOP key or when the CPU is in the wait state.

Ocassionally, on a virtual machine, execution may temporarily stop, perhaps because of a paging interrupt or a higher-priority job. This is the

virtual version of an operator on the real machine with jittery fingers on the STOP/GO keys on the system console.

To provide a sufficiently high-resolution clock on a virtual CPU, the CPU clock should be stored and reloaded as part of the hardware interrupt actions.

7. VIRTUAL PAGING

It is even possible to provide virtual paging hardware on the virtual machine, by a sufficiently complicated trick. This makes it possible to become recursive, and run the whole virtual operating system under itself on one of its own virtual machines. The trick works as follows. Some instructions must exist on the machine to enter paging mode, to indicate that a particular page map is to be used, or to alter the page table. These instructions must be trapped whenever the virtual machine attempts to execute them on the virtual page table.

The virtual machine monitor keeps two sets of page tables:

- the "formal" page table that says where the pages of the virtual machine reside, in storage or on disk, and
- the "actual" page table actually used by the paging hardware.

The actual page table must be the functional composite of the virtual page table and the formal page table. The formal and actual page tables will of course be the same whenever the virtual machine is not operating in virtual paging mode.

The actual page table must be continually updated to reflect changes in the formal or virtual page tables. Changes in the formal page table are performed by the virtual machine monitor itself; these are therefore fully known to it. When the virtual machine attempts to change its virtual page table, however, the attempt must be trapped. If changes are made using special instructions, this is easy; they need only be invalid in user mode. If, as is usual, page tables are changed simply by writing in them, memory protection must be invoked to trap all instructions trying to change the page table. They must then be laboriously interpreted by the virtual machine monitor.

One serious restriction on the composite paging tables is that the virtual pages have to fall within real pages; it is usually impossible to construct the composite page table if virtual pages fall across page bound-

aries. If the real machine itself does not follow this restrictions, there will be incompatibility between the real and virtual machines. Any self-virtualizing virtual machine monitor will have to follow this restriction voluntarily, even if the hardware does not impose it.

Another representation of storage maps should be mentioned, since it is a good deal simpler than a complete paging map. Originally conceived as a protection mechanism, it can be used only if the virtual memory need be no larger than the real memory. There need be only one page, which can be described by a "relocation-bounds register". The relocation-bounds register consists of two parts, a "relocation" register, whose value is added to every virtual address to obtain the real address, and a "bounds" register, which contains the upper, and sometimes lower, limits on valid virtual addresses. The virtual address space consists of a single contiguous block of addresses, and it is mapped onto a single continuous block of main memory. The great advantage of a relocation-bounds register is that composition of virtual storage mappings is trivial. The great disadvantage is that it does not provide a virtual address space larger than the real address space, it is relatively inflexible, and it does not provide sharing of storage between jobs. On the other hand, it may be possible to use relocation/bounds registers as well as paging in one system. The relocation/bounds registers can be used by the virtual machine monitor, and the paging provided by a conventional operating system operating under it.

8. RESOURCE MAPS

Let us now look at the ideas involved more abstractly.

When a process is running, it makes use of various virtual resources. There exists some mapping f mapping the virtual resources onto real ones. This mapping may be realized by hardware, or by a combination of hardware and software. If a machine is run without any virtualization, the mapping is the identity map

$$V \xrightarrow{f} R$$

If the virtual machine is functionally identical to the real machine, it is possible to run another copy of the virtual machine monitor on the virtual machine. This virtual machine monitor will create a set of virtual

virtual resources V_1 and map them onto the virtual resources V by some map f_1:

$$V_1 \xrightarrow{f_1} V \xrightarrow{f} R.$$

The composite mapping

$$f_1 \circ f: V_1 \rightarrow R$$

is then the effective resource map, through two layers of virtual machine monitor. Each virtual machine monitor gets to define a part of the resource translation. Unless special measures are taken, this can become quite inefficient when virtual machine recursion becomes deep:

- Special hardware to perform the mapping composition can eliminate much software intervention.
- Special software can maintain the functional composite $f_1 \circ f$ instead of or as well as the individual maps f_1 and f_2.

Although it must not be possible for a virtual machine monitor to detect whether it itself is running on a virtual or real machine, it is possible for it to know whether the virtual machine it creates is using the virtual virtualizing instructions, and to take appropriate evasive action with $f_1 \circ f$. Virtual machine monitors can in this way construct parts of the functional composite.

One problem in building a virtual system is that of representing the resource map(s) in an efficient manner that requires little software intervention for normal cases. It is also important to be able to map various kinds of resources onto each other, for example, virtual main store onto disk, or virtual disks onto main store.

The mapping function is usually required to take little more time, on the average, than it would take to access the resource directly. As a result, access to virtual main store must normally be done with approximately the speed of real main store. This requires the virtual store resource map to be to a large extent implemented in hardware.

Mapping of input and output devices is not as critical, because they are normally slow anyway. Even if software is used to perform the mapping, virtual disks mapped onto main store can easily be faster than real disks.

9. HIGH-LEVEL VIRTUALITY

There exist computer systems which are programmed only in high-level languages. There do not even exist assemblers for them. Such machines usually have special hardware for performing the various run-time checks (such as subscripts ranges) required for security in the high-level languages. The entire operating system on such a machine is written in a high-level language, which must have special provisions for operations not available to the every-day user. Normal users are prevented from using such sensitive operations not by hardware restrictions, but by preventing access to the compilers that generate the sensitive instructions. Files of object code are marked as such by officially registered compilers, and the operations of registering a compiler and marking object code are themselves sensitive. It is impossible for a user to sneak object programs containing forbidden instructions into the system by, for example, cunningly arranging exotic combinations of bits in a file. The protection mechanism is thus created by the compilers instead of by the hardware. Compilers must be considered in some sense analogous to microcode on other machines.

This approach yields some intriguing possibilities for safe tampering with variations on the operating system. The systems programming language compiler can be provided with an option - either actually to compile actual sensitive operations into the object code, or to compile them as calls to a run-time system which carefully updates a model of the real system. Such a virtual machine has many of the aspects of a normal virtual machine, except that it is not bit-compatible. But why should it be bit-compatible? Since the hardware is approachable only via high-level languages, it suffices for it to be completely high-level language compatible.

On conventional machines, so-called "student-batch compilers" often provide such virtuality, by separating the students' programs from the real operating system file system for reasons of security or efficiency. This also allows limits on the amount of input or output to be enforced independently of the real operating system.

The serious drawback of this kind of architecture is that is as dangerous to write a new compiler for the system as it is to write new microcode for a conventional one. A new compiler begins its life by generating incorrect object code. In the absence of complete hardware security measures, and without the use of a standard assembler for compilers which guarantees security by checking source code semantically, this is dangerous. New compilers

are therefore usually debugged during off-hours when there are no other users on the system, but the introduction of the (nearly) debugged compiler is still a risky business. If it were possible to make such a machine fully virtualizable, it would much simplify the installation of new compilers. The virtuality could provide the necessary security, and might well become a normal part of the protection architecture.

10. IMPLEMENTATION TECHNIQUES

The normal method of creating complete virtual machines requires a distinction between a "user" mode, in which sensitive instructions are trapped, and a "system" mode, in which they are executed. If a virtual machine (which operates in real user mode) attempts to execute a sensitive instruction, the virtual machine monitor (in system mode) receives an interrupt. If the virtual machine was in virtual user mode, the interrupt is passed back to it; if in virtual system mode, the instruction is interpreted.

It is possible to create virtual machines on systems which behave slightly differently. For example, on many minicomputers, sensitive instructions do not cause traps in user mode, but instead are executed as no-operation instructions. On such a system, the virtual user mode will automatically work correctly, but the virtual system mode will require instruction-by-instruction interpretation of all instructions. Such a system is practical if the virtual machine only rarely enters system mode.

It is also possible to have hardware that can understand the resource map without software intervention (at least, in the vast majority of cases). It may be quite feasible to have it interpret relocation tables for disk addresses and input/output devices as well as just a paging table for storage. Indeed, on some machines, input/output devices are addressed in the same way as storage, and can therefore use the same storage protection mechanism. Writing to certain addresses in store may be interpreted as requesting the disk drive, for example, to perform some action. To prevent or control disk access, one prevents or controls storage access.

If the map is to be implemented in hardware, it is advantageous to be able to handle map composition by hardware as well, otherwise there will be considerable overhead when a virtual machine is run under itself. Composition can easily be performed by letting the machine look up its virtual resource in table after table, serially, until the final real resource is

discovered. In principle, of course, each level of translation may require software intervention if the map can not be done completely in hardware.

The "hardware virtualizer" will then contain a *stack*. This stack is not available for examination by virtual (or real) machines. Each stack entry is an address or other code identifying a resource map. When a program wishes to create a new virtual machine with a given resource map, it executes an instruction to place another entry onto the stack. When, during execution on a virtual machine, one of the resource maps fails (perhaps because of a missing page), the trap is given to that virtual machine which placed the failing resource map on the stack. The only difference between a virtual machine and a real machine is that the resource map stack is empty; no translation takes place. Of course, since the stack is not available for examination, the real machine has no way to know this; and thus does not discover that it is real.

It is possible, of course, that this stack becomes full. By counting the number of times it is possible to place new entries onto the virtualizing stack, it might be possible to find out how deeply nested one already was in virtual machine nesting. To prevent this, there must be some software handling to perform one more level of map translation after the stack fills up. If the depth of the hardware virtualizing stack is n, each virtual machine will then handle the stack overflows caused by the virtual machine n levels deeper.

11. VIRTUAL DATA

So far we have concentrated on virtual hardware, such as disks, main storage, and CPU's. It is also possible to have virtual data. Once again, we shall try to make the hardware do the common operations very rapidly by itself, and resort to software only when the hardware is inadequate or excessively complex.

Suppose, for example, that we wish to perform calculations on arbitrarily large integers. Remember, on a typical machine, the typical programmer rarely *knows* that his integers will remain less than 2^{16} or 2^{47} or whatever the local limit is, but he simply hopes it. Rather than requiring him to go to all the trouble of analysing his program to determine this, it would be much more convenient if the programming system automatically went over to another mode when required.

Part of the efficiency of conventionally represented integers on conventional machines is that they take a fixed small amount of space. The vast majority of integers needed are, in fact, small enough to be presented in this fixed amount of space. This means that they can efficiently be placed in structures, arrays, and other data structures with a relatively inflexible mapping function. How can we provide arbitrary precision without incurring large overheads when it is not necessary?

In order to preserve the fixed size property in some sense when we work with arbitrary precision, we shall have to use pointers. A pointer of known fixed size can point to data of unknown size.

The new-style integer looks as follows:

- a bit which tells which representation is used, and
- if it is possible, the integer represented in the conventional way, but
- otherwise, a pointer to some storage area sufficiently large for the integer.

The bit we shall call the "exception bit", or a "tag bit".

The exception bit itself need be processed by the hardware only insofar that if it is on, a trap to a user routine takes place; if it is off, the machine can simply go on with the calculation. The user routine must be capable of removing the tag bit in order to process the rest of the word, and it must also be capable of finding out which machine operation was attempted.

It is probably best to leave the arbitrary-size operations to software, since the necessary storage allocation must be done in a manner consistent with the rest of the programming system - a matter about which the operating system should know nothing.

Exception bits are a relatively simple but flexible feature for hardware. It is even possible to have several tag bits, some combinations of which can be interpreted by hardware and others left to software. It may also be useful to distinguish between "system" tags and "user" tags. The former can be created and removed only in system mode; the latter in user mode as well. By using system tags, an operating system can hand some tags of system control data to the user, to be copied and stored as the user chooses.

If this data finally comes back as an operand in a system call, the system can nonetheless be certain that it has not been tampered with. Such a trick

might be used to avoid a limit on the depth of the virtual machine stack in section 10.

12. APPLICATIONS

It should by now be clear that we have the technical means to realize virtually any kind of virtual machine on most conventional hardware. For efficiency, the virtual machines usually have an instruction set very similar to that of the real machine, but this is not absolutely necessary. Microprogramming is one good way to realize virtual machines that are quite different from the underlying real machine, if any.

The virtual machine need not:

- have the same instruction set,
- have the same I/O configuration,
- have any of the same I/O devices,
- have the same amount of memory, or
- have either more or less of any resource

as the underlying real machine.

The technical limitations on the creation of virtual machines are not the same as those on the creation of "real" machines. The question arises:

"What kind of virtual machine do we want to have?"

13. FILES AND STORAGE

Let us compare random-access files with large data structures in main storage. There is no conceptual difference between them. Each contains many data which may have to be accessed in arbitrary order. Complicated buffering routines accomplish little for such files that paging does not do for main store. Furthermore, it often happens that some of a job's files are accessed more often than some of its main storage. It is a fairly natural step to treat files and storage as equivalent. Opening a file becomes adding the file to one's address space. Access to the file can then be made using normal main storage access, supplemented by paging. The main difference between opening such a file and requesting storage is that the initial contents may be important.

To make it practical to include files in the virtual address space, the virtual address space must be large. Real-life data bases can reach 2^{30} to 2^{40} bits; for the sake of expansion and flexibility, it may well be wise to set the addressing capacity a few orders of magnitude above this. A 48-bit address may suffice for another decade.

Such an address space would not be financially feasible without paging; MOS storage may be getting cheap, but not *that* cheap.

It is not necessary that all of this address space be associated with real storage for it to be worthwhile. It may be useful to be able to claim large blocks of virtual address space just to ensure that later, if the storage is needed, contiguous address space will be available despite fragmentation.

On some machines it is possible to write programs in such a way that they contain no absolute addresses. If any addresses are needed at run time they can be generated using base registers or other methods. If the file system is implemented as virtual storage, all that is needed to load a program is to include the file containing it in the job's virtual storage by altering the page tables. There is no reason whatsoever why the same copy of the same program should not simultaneously be part of another process's virtual storage, possibly even at a different virtual address. A single page can in principle even occur more than once in the virtual address space of a single process, if this is desired.

Programs, if so shared, would typically be kept read-only. It is, on the other hand, possible to use writable shared storage as a means of communication between processes.

It should be noted that the total size of all files in the system can greatly exceed the available address space. A file need be in the address space of a process only while it is being used.

14. PROTECTION

Hardware on which one can efficiently create virtual machines can provide a basis for flexible and secure protection mechanisms. The basic necessary mechanism is present - the ability to monitor and restrict any and all communication between a process and the outside world. In third generation systems this was usually thought of as protecting the system against the users, and protecting the users against each other; the system

was considered trustworthy. Experience shows that this optimism was misplaced. A typical third-generation operating system consists of enormous amounts of code, some old, and some new. It is unreasonable to think that such a system will ever be debugged; new sections sometimes fail, and in so doing they may mutilate the old parts sufficiently that the whole operating system is brought down. It is becoming clear that the components of an operating system must themselves also be protected against each other, and that the user must be protected against those he does not use.

One first attempt is to separate multiprogramming from the rest of the system. The kernel of such a system consists of a virtual machine monitor, whose purpose is the creation of many virtual machines.

Into each virtual machine a copy of an operating system can be loaded, and each user then gets his own operating system. The virtual machine mechanism separates users from each other; each can be hurt only by the misbehaviour of his own virtual operating system, and not by others.

This approach can significantly improve the availability of a computer system, but it has drawbacks of its own.

- The virtual operating system itself can still fail just as always, possible causing damage to user data.
- The virtual machine monitor may also be complex and have the same failing as the former operating system.
- The decision how to restart the virtual operating system after a crash is in the hands of unschooled users, instead of experienced operators. This may lead to dangerous restarts in dangerous situations.

Each component of an operating system needs certain "capabilities" in order to do its work. A file system manager needs to be able to read and write the file catalog, but need do nothing with files; the routines that perform input/output need to be able to read and write on specific files, but do not need a generalized capability to read or write on all files in the system. Each component of a system (and each user) should therefore be as isolated as a virtual machine, except that it will be permitted certain external actions upon explicit request. Performing such an action is "exercising" a "capability". Typical capabilities may be to read or write from parts of main storage, to perform actual input or output operations, to call some program or communicate with some process which may operate with a different set of capabilities, or even to request new capabilities. Successful opening of an output file, for example, endows one with the capa-

bility of calling input/output routines that write on the file.

There is at present much experimentation and controversy about the precise form that capabilities should take, about the way that they can be created, modified, and passed from one program to another, and even whether they should belong to programs, processes, or some mixture of the two.

15. CONCLUSION

And they lived happily ever after [1].

REFERENCE

[1] *Workshop on virtual Computer Systems*, 26-27 March 1973, ACM SIGARCH-SIGOPS.

MATHEMATICAL CENTRE SYLLABUS 32, 1976, III, 43-60

ENIGE ASPECTEN VAN HET BURROUGHS B6700 SYSTEEM

H. ROUMEN
Technische Hogeschool, Eindhoven

0. INLEIDING

Op het rekencentrum van de Technische Hogeschool Eindhoven is een Burroughs B6700 computer geïnstalleerd bestaande uit de volgende componenten:

2 processoren
256K woorden geheugen (1 woord = 48 informatie bits)
1 I/O processor
1 datacommunicatie processor met 8K woorden lokaal geheugen
100 Mbytes HPT-disk
8 disk pack units
6 tape units
de gebruikelijke langzame randapparatuur

Aan de datacommunicatieprocessor (DCP) zijn een aantal stations gekoppeld, te weten teletypes, remote job entries, minicomputers (PDP 9, PDP 11/45, PDP 11/20, PDP 11/10, P855) en plotters.

Het operating system van de B6700 (Master Control Program) is een multiprocessor/multiprogrammeringssysteem en is geschreven in de hogere programmertaal ESPOL. Speciaal voor het afhandelen van het datacommunicatieverkeer zijn een aantal programmadelen (Message Control Systems) ontwikkeld die in feite als een uitbreiding van het operating system beschouwd kunnen worden. Deze Message Control Systems zijn geschreven in een speciaal daarvoor ontwikkelde taal, DCALGOL. Voor de gebruikers staan als programmeertalen BEA (Burroughs Extended Algol), BEATHE (Burroughs Extended Algol THE versie), Fortran, Cobol, PL/I, en Basic ter beschikking.

In deze syllabus zal allereerst in het kort ingegaan worden op de architectuur van de B6700. Vervolgens zullen in iets meer detail enkele facetten van het operating system (MCP) bekeken worden, namelijk het geheugenbeheer, de scheduling en de protectie van files. In paragraaf 5 zal aan-

dacht besteed worden aan de software die te maken heeft met de datacommunicatie-aktiviteiten. Aan het einde van deze syllabus wordt nog een korte verhandeling gegeven over de programmeertalen ESPOL en BEA.

Alle in deze syllabus gegeven eigenschappen van het systeem reflecteren de II.6 software release van Burroughs.

1. MACHINE-ARCHITECTUUR

1.1. *Stacks*

Aan ieder proces binnen het B6700 systeem wordt een stack toegekend. In de stack worden opgeslagen:

- de waarden van de variabelen uit het proces,
- verwijzingen naar data segmenten,
- statusinformatie van het proces,
- enkele controle woorden.

Een van deze controle woorden is het Mark Stack Control Word (MSCW). Bij elke blockingang en bij elke procedure aanroep wordt een MSCW op de stack geplaatst. Bij het verlaten van de procedure of van het block wordt het overeenkomstige MSCW weer van de stack verwijderd. Elk MSCW bevat twee velden, die verwijzingen bevatten naar vorige MSCW's in de stack:

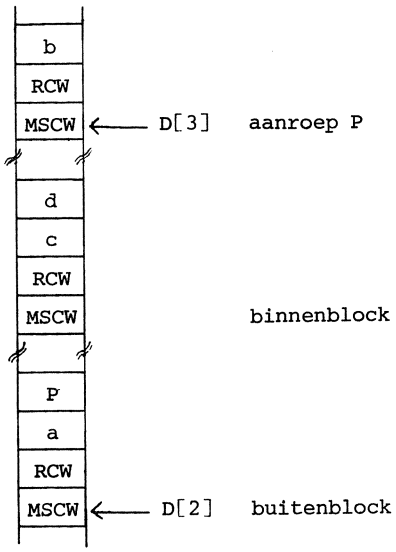
- een verwijzing naar het voorafgaande MSCW (dynamische link);
- een verwijzing naar het laatste in de stack geplaatste MSCW dat een lexicografisch niveau heeft, dat 1 lager is dan het lexicografisch niveau van dit MSCW (statische link).

Het lexicografisch niveau wordt door de compiler bepaald. Bij het begin van de vertaling van een procedure of block wordt het niveau met 1 verhoogd; bij het einde van een block of procedure weer met 1 verlaagd. De beginwaarde voor het buitenblock is gelijk aan 2. Op elk moment vormen de MSCW's die met elkaar verbonden zijn via de statistische link de *statistische lijst*; op analoge wijze kunnen we spreken van de *dynamische lijst*.

Elke processor bevat 32 zogenaamde display registers. Voor een aktief proces bevatten een aantal van deze registers de adressen van de MSCW's in de stack, die op dat moment de statistische lijst vormen (zie fig. 1).

```
begin integer a;
      procedure P;
      begin integer b;
            .
            .
            b := 3;
            .
            .
      end;

      begin integer c,d;
            .
            .
            P;
            .
            .
      end
end
```



stackbeeld, net voordat de statement b := 3 *wordt uitgevoerd.*
(RCW = Return Control Word)

fig. 1

1.2. *Adressering*

Adressering van variabelen binnen de stack vind meestal plaats relatief ten opzichte van een MSCW. De compiler kent aan iedere variabele een adreskoppel (*lexicografisch niveau, displacement*) toe.
Met behulp van het lexicografisch niveau wordt een display register geselecteerd; de displacement geeft het aantal posities boven het bijbehorende MSCW aan. Zo hebben de variabelen uit fig. 1 als adreskoppels:

a = (2,2); b =(3,2); c =(3,2); d =(3,3).

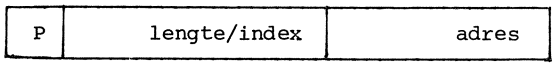
(De beginwaarde voor de displacement is gelijk aan 2).
Hoewel de variabelen b en c in dit voorbeeld gelijke adreskoppels hebben, zal indien gerefereerd wordt aan (3,2) hiervoor op dit moment de stackpositie voor b genomen worden. Dit omdat op elk moment de rij van display registers de directe adresseringsomgeving aangeeft.
Bij blockverlating en procedure-uitgang zullen de inhouden van de display registers eventueel aangepast moeten worden om de nieuwe adresseringsomgeving te reflecteren. Hoe dit op een vrij eenvoudige wijze kan, wordt beschreven

door ORGANICK [1].

Soms is het noodzakelijk om buiten de directe adresseringsomgeving te adresseren (parameters bij een procedure) of zelfs in een andere stack (parallelle processen). In die gevallen wordt relatief ten opzichte van het begin van de desbetreffende stack geadresseerd.

1.3. *Datasegmenten*

Voor een array wordt 1 stackplaats gereserveerd, die gevuld wordt met een descriptor met de volgende layout:



P = presence bit

fig. 2

Op het adres zoals aangegeven in de descriptor begint een segment van de in de descriptor aangegeven lengte. Dit segment bevat dan de actuele data of op zijn beurt weer descriptoren die verder verwijzen naar andere segmenten. Dit laatste is onder andere het geval bij meer-dimensionale arrays. We noemen een segment gevuld met descriptoren een *dopevector*. Een dopevector wordt ook gegenereerd voor array-rijen waarvan het aantal elementen groter dan 1023 is. In dat geval worden voor zo'n rij segmenten opgebouwd met een lengte van 256 woorden, waarnaar verwezen wordt met behulp van een extra dopevector. Een voorbeeld van een dergelijke gesegmenteerd array is gegeven in fig. 3.

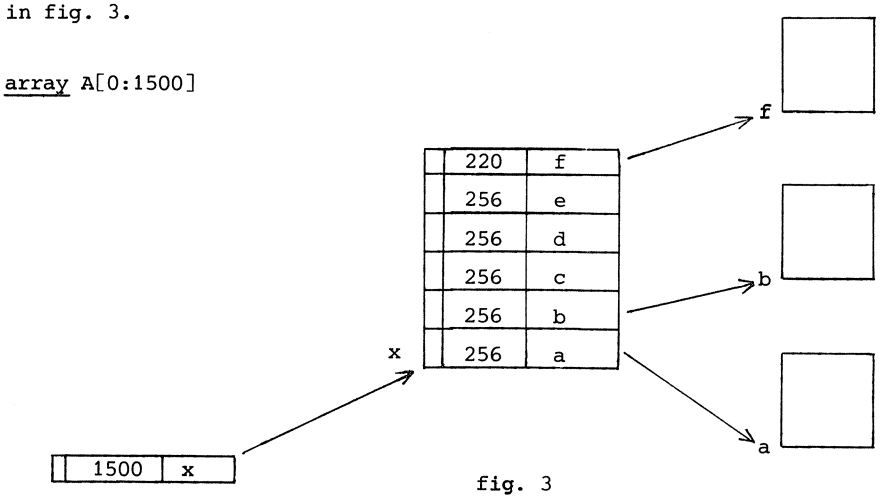


fig. 3

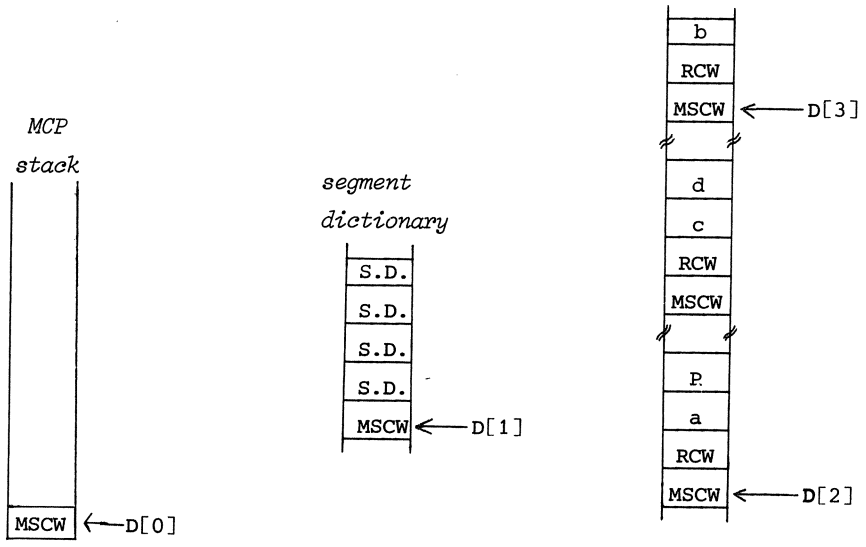
Het presence bit geeft aan waar het bijbehorende segment gevonden kan worden. Afhankelijk van de waarde van P stelt het in de descriptor gegeven adres een kerngeheugenadres voor of een diskadres.

Om een element uit een segment te kunnen selecteren, wordt een copy van de descriptor op de stack geplaatst, waarbij het lengteveld gevuld wordt met de index en waarbij een bit in de descriptor anageeft dat dit een geïndexeerde descriptor is. Onder bepaalde omstandigheden wordt ook nog het adresveld in de nieuwe descriptor gevuld met het adres van de oorspronkelijke descriptor. Voor een beschrijving hiervan wordt verwezen naar [2]. Voor gesegmenteerde arrays (herkenbaar aan een speciaal bit in de descriptor) wordt eerst met behulp van de expressie *index div 256* een descriptor geselecteerd uit de dopevector; vervolgens vindt er een indexoperatie plaats op deze descriptor met de waarde van *index mod 256*.

1.4. *Codesegmenten*

De objectcode zoals deze door de compiler gegenereerd wordt, wordt opgeslagen in segmenten van variabele lengte. Een segment bevat draabij de code voor een block of voor een procedure (in geval van Algol-programma's). Naar een codesegment wordt verwezen door een segment descriptor die een zelfde layout heeft als de descriptor gegeven in fig. 2.
Alle segment descriptoren voor een programma worden verzameld in een aparte stack, de *segment dictionary* geheten. In de segment dictionary worden verder ook de descriptoren geplaatst voor de read-only arrays van het programma. De segment dictionary bevat dan ook alleen maar statistische gegevens.

Voor een actief proces verwijst het D[1] register naar het begin van de bij dit proces behorende segment dictionary (op deze plaats staat weer een MSCW). De koppeling van de actieve stack van het proces (de werkstack of D[2] stack) met de bijbehorende segment dictionary vindt plaats met behulp van de statische link uit het MSCW zoals aangewezen door het D[2] register. Op een zelfde wijze vindt vanuit de segment dictionary een koppeling plaats met de stack van het MCP, waarnaar altijd het D[0] register verwijst. Een en ander is schematisch weergegeven in fig. 4.



S.D. = Segment Descriptor
RCW = Return Control Word

fig. 4

Voor elke procedure en voor elk block wordt in de stack een woord geplaatst, dat de volgende gegevens bevat:

- index van de bijbehorende segment descriptor in de D[1] of de D[0] stack;
- het beginadres in het door de segment descriptor aangewezen segment.

Een dergelijk woord wordt Program Control Word genoemd. Het Program Control Word voor het buitenblock is opgenomen in de segment dictionary.

Re-entrancy kan nu eenvoudig worden gerealiseerd door bij het opstarten van een proces na te gaan of deze code reeds gebruikt wordt. Is dit het geval dan wordt een nieuwe werkstack opgezet en wordt met behulp van het D[2] register van deze stack een koppeling aangebracht met de reeds bestaande segment dictionary. In de segment dictionary is een speciaal woord aanwezig dat aangeeft hoeveel processen van deze segment dictionary gebruik maken. Bij het einde van een proces wordt de segment dictionary pas verwijderd, als dit aantal, na aflaging met 1, gelijk aan 0 is geworden.

Voor een verdere en meer gedetailleerde beschouwing wordt verwezen naar de artikelen van CLEARY [3] en van HAUCK & DENT [4].

2. GEHEUGENBEHEER

Het geheugen dat door een proces bezet wordt kan onderverdeeld worden in 2 klassen:

- save geheugen, en
- overlayable geheugen.

Tot het overlayable geheugen worden die segmenten gerekend, die zowel in het kerngeheugen als op disk mogen staan. Het save geheugen moet permanent (voor de duur van het proces) in de kernen aanwezig zijn.
Enkele voorbeelden van save geheugen zijn: de stack, de segment dictionary, dopevectoren en input/output buffers.

2.1. *De SAVE en de OLAY lijst*

Het totale geheugen kan op elk moment beschouwd worden als een aantal segmenten van variabele lengte, die of in gebruik (in-use) of vrij (available) zijn.
Elk segment wordt voorafgegaan en direct gevolgd door memory links.
Voor segmenten die in gebruik zijn bevatten de memory links onder andere de volgende gegevens:

- stack nummer van het proces dat dit segment gebruikt;
- omvang van het segment;
- het adres van de descriptor die naar dit segment verwijst;
- informatie of het een save of een overlayable segment is.

Voor vrije segmenten bevatten de memory links onder andere:

- de omvang van het segment;
- informatie over de bruikbaarheid van dit segment;
- verwijzingen naar andere vrije segmenten.

Met behulp van deze verwijzingen worden twee lijsten opgebouwd, de SAVE lijst en de OLAY lijst. Een segment behoort tot de SAVE lijst als het fysisch ervoor liggende segment in gebruik en save is. Een segment behoort tot de OLAY lijst als de fysisch aangrenzende segmenten niet beide in gebruik en save zijn. Het zal duidelijk zijn dat bepaalde segmenten zowel in de OLAY lijst als in de SAVE lijst kunnen voorkomen. De lijsten zijn gesorteerd in opklimmende grootte van segmenten.

Wordt nu door een proces een segment aangevraagd, dan wordt als het

om een save segment gaat de SAVE lijst afgezocht en anders de OLAY lijst. Er wordt gezocht naar het kleinste segment in de lijst dat groot genoeg is om dit nieuwe segment te bevatten. Het eventuele restant wordt als dit groter is dan 15 woorden weer als nieuw segment in een van beide lijsten opgenomen. Is het restant kleiner dan wordt dit aan het nieuw aangevraagde segment toegevoegd, daarbij maatregelen nemend dat de aanvrager geen gebruik kan maken van het toegevoegde restant. Dit alles wordt gedaan om checkerboarding te minimaliseren. Wordt er geen voldoende grote ruimte gevonden in de betreffende lijst, dan moet er een overlay plaatsvinden van een of meerdere reeds in het geheugen aanwezige segmenten naar disk. We spreken dan van demand overlay. Daarnaast kent Burroughs nog een andere overlay strategie: de forced overlay, die in combinatie met demand overlay gebruikt kan worden. Of dit gebeurt kan met een operateursboodschap bepaald worden.

2.2. *Demand overlay*

Demand overlay treedt dus op als er bij aanvraag van geheugenruimte geen voldoende vrije ruimte wordt gevonden in de doorzochte lijst. Het MCP zoekt in het geheugen naar een aaneensluitend stuk, dat bestaat uit vrije segmenten en in gebruik zijnde overlayable segmenten en dat groot genoeg is om aan de aanvraag te kunnen voldoen. Het zoeken in het geheugen vindt cyclisch plaats; om de positie bij te houden waar men gebleven is, wordt een wijzer gebruikt: de *leftoff pointer*. Van de in de gevonden ruimte aanwezige overlayable segmenten wordt eerst geprobeerd om deze elders in het geheugen te plaatsen (core to core overlay). Lukt dit niet, dan vindt een overlay plaats naar disk.

Voor read-only segmenten (zoals codesegmenten en datasegmenten voor read-only arrays) hoeft geen I/O transport plaats te vinden; het enige dat moet gebeuren is het aanpassen van de bijbehorende descriptor. We spreken in een dergelijk geval van core to limbo overlay. Dat een segment read-only is wordt aangegeven in de memory links voor dit segment.

2.3. *Forced overlay*

Er bestaan twee systeem parameters, die de operateur dynamisch een waarde kan geven en die bepalen in welke mate geforceerde overlay plaatsvindt. De eerste parameter is *olaygoal* en geeft globaal aan welk percentage van het door een proces gebruikte overlayable geheugen per minuut vrij ge-

geven moet worden. De tweede parameter is *availmin* en geeft aan welk percentage van het totale geheugen ten alle tijde ter beschikking moet staan van het MCP (in feite is dit een stuk geheugen dat het MCP extra kan claimen en staat dan ook los van het actuele geheugengebruik van het MCP).

Elke 3 seconden wordt er door het MCP een onafhankelijk proces gestart, *WSSHERRIFF* genaamd, dat allereerst berekent hoeveel er voor ieder proces vrij gegeven moet worden en verder er voor zorgt, dat minimaal het berekende bedrag aan ruimte vrij komt. De berekening van het per proces vrij te maken percentage vindt plaats aan de hand van de formule:

$$\frac{100 - prioriteit}{50} \times \frac{olaygoal}{20}$$

Hierbij dient opgemerkt te worden dat de prioriteit van een proces kan variëren van 1 (laagste) tot 99 (hoogste). Van processen met een hoge prioriteit wordt weinig ruimte vrijgegeven; van processen met een lage prioriteit wordt veel geheugen vrijgegeven.

Voor het zoeken naar segmenten die verwijderd kunnen worden, wordt weer gebruik gemaakt van de left-off pointer.

Voor de overlayable geheugenruimte van het MCP zelf wordt dezelfde strategie gevolgd; wordt echter het totale overlayable geheugen van het MCP groter dan een bepaalde drempelwaarde (in het standaard Burroughs systeem 25000 woorden), dan vindt een sterk vergrote overlay plaats voor het MCP. Het te verwijderen percentage wordt dan gelijk gemaakt aan 25% (van het totale overlayable geheugen).
Opgemerkt dient nog te worden dat met behulp van een operateurs boodschap per proces afzonderlijk een olaygoal ingesteld kan worden.

Een volgende taak van WSSHERRIFF is het handhaven van de extra geheugenruimte voor het operating system zoals aangegeven door de parameter availmin. Op het moment dat WSSHERRIFF ontdekt dat het totale vrije geheugen kleiner is dan het percentage zoals aangegeven door availmin, wordt het proces met de laagste prioriteit onderbroken. Vervolgens wordt het totale door dit proces bezette overlayable geheugen vrijgemaakt. Het onderbreken vindt alleen maar plaats als het aantal processen binnen het systeem groter is dan het aantal processoren.
Een onderbroken proces kan in een volgende cyclus van WSSHERRIFF weer actief gemaakt worden. Het criterium hiervoor is dat het beschikbare vrije geheugen groter is dan het gemiddelde geheugengebruik van dit proces tot nu toe of dat het aantal processen binnen het systeem kleiner is geworden dan het aan-

tal processoren.
Tot slot zij opgemerkt dat als forced overlay gebruikt wordt, ook demand overlay plaats zal kunnen hebben.

2.4. *SWAPPER*

Naast de in 2.2 en 2.3 beschreven technieken is er binnen het B6700 systeem nog een derde mogelijkheid, die parallel aan en onafhankelijk van beide andere mogelijkheden gebruikt kan worden. Er bestaat namelijk een mogelijkheid om het geheugenbeheer en het processorbeheer voor bepaalde processen te laten besturen door een onafhankelijk proces, SWAPPER genaamd. Dit proces kan met behulp van een enkele operateursboodschap opgestart worden. Als SWAPPER actief is staan automatisch alle interactieve programma's en verder een bepaalde klasse van batch programma's (zie 3.1.) onder diens controle. Voor een actief proces wordt al het overlayable geheugen geplaatst in een aaneengesloten stuk geheugen (altijd een veelvoud van 990 woorden). Wordt het proces inactief, dan wordt dit overlayable geheugen in zijn totaliteit naar disk getransporteerd.

Aan ieder proces wordt door SWAPPER gedurend een bepaalde timeslice, die per proces berekend wordt, de processor toegelaten. Naast de prioriteit is ook het geheugengebruik van het proces mede bepalend voor de grootte van de timeslice (de timeslice is groter voor processen met een groter geheugengebruik).

Metingen met SWAPPER hebben aangetoond dat vooral wat de turnaround tijd voor interactieve processen betreft, voordelen te behalen zijn. Echter bij het opstarten van SWAPPER moet als parameter het maximale geheugen opgegeven worden dat een proces onder controle van SWAPPER mag gebruiken. Deze ruimte wordt permanent door het systeem gereserveerd voor SWAPPER en is dan ook niet toegankelijk voor parallel lopende batch processen. Een zelfde opmerking kan gemaakt worden voor de ruimte op disk. Een goed geheugengebruik (en ook diskgebruik) bij het gebruik van SWAPPER kan dan ook alleen gegarandeerd worden bij een voldoende aanbod van programma's die onder controle van SWAPPER dienen te verlopen.

Tot slot zij nog opgemerkt dat SWAPPER geen bemoeienis heeft met de werkelijke processortoekenning. Dit blijft gebeuren door de daarvoor bestemde algorithmen binnen het MCP. In paragraaf 4 wordt hier nader op ingegaan.

Voor meer informatie over het geheugenbeheer wordt verwezen naar de door Burroughs uitgebrachte manual System Miscellanea [5].

3. JOB EN TASK SCHEDULING

Alvorens we ingaan op het processorbeheer, zullen we even aandacht besteden aan het afhandelen van een job door het systeem.

3.1. *WFL compiler en CONTROLLER*

Als een job in een kaartlezer geplaatst wordt, wordt door het MCP een onafhankelijk proces opgestart, Work Flow Language (WFL) compiler geheten. De WFL-compiler interpreteert en genereert code voor de job control kaarten en plaatst deze tezamen met de bij de job behorende data-decks in een file op disk. Bij het afsluiten van deze plaatst het MCP een boodschap in de input queue voor de CONTROLLER. De CONTROLLER is in feite een routine binnen het MCP die regelmatig geactiveerd wordt.
Als reactie op deze boodschap wordt de aangeboden job door de CONTROLLER in een input queue voor het systeem geplaatst. Welke input queue dat is, wordt door de gebruiker in het job control programma vastgelegd.
De keuze van de input queue bepaalt onder andere:

- de maximale CPU- en IO-tijd
- het gebruik van randapparatuur
- de prioriteit
- of de processen in swapruimte verwerkt moeten worden of niet.

Bovendien is per input queue bepaald hoeveel processen afkomstig uit deze queue tegelijkertijd actief mogen zijn (de mixlimit). Een en ander is dynamisch instelbaar.
De CONTROLLER selecteert vervolgens (als de mixlimits dit toelaten) een job en geeft een melding aan het MCP dat de eerste task van deze job gestart kan worden.

3.2. *Processorbeheer*

Het systeem handhaaft voor de scheduling van processen twee queues: de *sheet queue* en de *ready queue*.
In de sheet queue bevinden zich de tasks waarvan het verwachte geheugengebruik groter is dan het op een bepaald moment beschikbare geheugen; in de ready queue bevinden zich processen die klaar staan om de processor te ontvangen.
De geheugenschatting voor een proces wordt in eerste instantie door de compiler bepaald; is dit proces reeds eerder uitgevoerd, dan wordt als geheugenschatting het gemiddelde geheugengebruik van de voorafgaande keren gebruikt. De geheugenschatting van de compiler voor een proces is de som van de volgende termen:

- stack estimate.
  De benodigde stackruimte voor het buitenblock plus de stackruimte op het volgende lexicografisch niveau.
- array estimate (de som van de schattingen voor de grootste 10 arrays).
  Voor een array $A[l_0:u_0,\ldots\ldots,l_{m-1}:u_{m-1}]$ is de schatting:

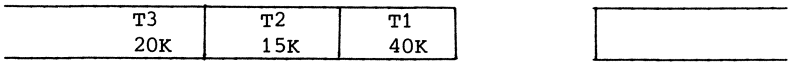
  $$\sum_{i=0}^{m-1} size_i$$ waarbij

  $size_i = u_i - l_i + 1$ voor constante grenzen en
  $size_i = 100$ voor dynamische grenzen.
- file buffer estimate (voor de 4 grootste files)
- totale code segment grootte
- maxsortsize
  Een schatting voor het geheugengebruik indien een bepaalde sorteerprocedure wordt aangeroepen.
- de constante 512.

Voor de scheduling wordt de geheugenschatting vergeleken met *available core - schedcore*. Is de geheugenschatting groter, dan wordt de task in de sheet queue geplaatst en anders in de ready queue. De volgorde van de tasks in de sheet queue is op basis van de prioriteit (zie later). Regelmatig wordt de sheetqueue geïnspecteerd om als er voldoende ruimte is (available core - schedcore) een of meerdere tasks te transporteren naar de ready queue.

Schedcore is een grootheid die ingevoerd is in het MCP om te voorkomen dat direct na elkaar tasks in de ready queue geplaatst worden. Dit zal toe-

gelicht worden aan de hand van het volgende voorbeeld. Stel er staan 3 tasks in de sheet queue met als geheugenschatting 40K, 15K en 20K en er is op het moment van beschouwing 50K aan beschikbaar geheugen (zie figuur 5);



sheet queue ready queue

fig. 5

Wordt nu task T1 van de sheet queue naar de ready queue verplaatst, dan zou bij een volgende inspectie van de sheet queue de mogelijkheid bestaan dan T1 slechts een kleine fractie van de geschatte 40% gebruikt en dientengevolge zouden T2 en eventueel ook T3 in de ready queue kunnen komen, met kans op een te grote aanvraag voor geheugen (overlay).

Vandaar dat bij het transporteren van een task naar de ready queue de waarde van schedcore verhoogd wordt met de geheugenschatting van deze task. Elke seconde wordt een routine van het MCP uitgevoerd, die schedcore verlaagt met een factor 0.707 en die nagaat of er tasks in de sheet queue staan die naar de ready queue kunnen.

In de queues staan dus de processen gerangschikt op basis van prioriteit. Globaal gezien is de prioriteitsvolgorde van hoog naar laag:

- processen en procedures van het MCP (zoals SWAPPER, WSSHERRIFF)
- Message Control Systems (zie paragraaf 5)
- Control Programs (tot deze categorie behoren onder andere de processen die verlopen onder besturing van SWAPPER; Control Programs kunnen nooit door WSSHERRIFF onderbroken worden)
- gebruikersprogramma's.

Binnen elk van deze categorieën zijn de processen in prioriteitsklassen gerangschikt, waarbij de prioriteitsklasse bepaald wordt door de gebruiker of door de input queue van het proces.
Binnen een prioriteitsklasse zijn de processen gerangschikt volgens de expressie:

$$\frac{CPU\text{-}tijd}{CPU\text{-}tijd + IO\text{-}Tijd}$$

(als *CPU-tijd + IO-tijd = 0*, dan wordt hiervoor de geheugenschatting van het proces genomen)

Hoe lager de waarde van deze expressie, hoe hoger de prioriteit, wat inhoudt, dat de IO gebonden processen een hogere prioriteit hebben dan de CPU gebonden processen.

4. PROTECTIE

Iedere gebruiker van het B6700-systeem op de THE heeft een usercode die als volgt opgebouwd is:

U < gebruikersnummer > S < sponsornummer >

De gebruiker is verplicht deze usercode te vermelden in het job control programma en bij het inloggen voor een interactieve sessie. Aan elke usercode kunnen één of meerdere passwords verbonden worden. Het maximale aantal wordt bij creatie van de usercode bepaald; tevens wordt dan vastgelegd of voor deze usercode interactief gebruik van het systeem toegestaan is.

De usercode speelt onder andere een rol bij het protectiemechanisme van diskpack files.
Aan elke file binnen het B6700-systeem zijn een aantal attributen gekoppeld. Twee van deze attributen bepalen de protectie van de betreffende file, te weten SECURITYTYPE en SECURITYUSE. De mogelijke waarden voor SECURITYTYPE zijn:

- PRIVATE Alleen de creator heeft toegang tot de file; dit is de default waarde.
- CLASSA De file is toegankelijk voor alle andere gebruikers.
- CLASSB De file is toegankelijk voor bepaalde gebruikers. Welke gebruikers dit zijn en welk gebruik ze van de file mogen maken moet vastgelegd worden in een speciale file, GUARDFILE geheten.

Het attribuut SECURITYUSE bepaalt, welk gebruik van de file gemaakt mag worden indien SECURITYTYPE als waarde CLASSA heeft. De mogelijke waarden voor dit attribuut zijn:

- SECURED Alleen de creator heeft toegang met als uitzondering dat als dit een codefile is, deze uitgevoerd mag worden.
- IN Read-only file.
- OUT Write-only file.
- IO Zowel in- als uitvoer toegestaan (default waarde).

De waarden van deze attributen kunnen bij file declaratie meegegeven worden en kunnen verder door de creator van de file met behulp van een job control statement veranderd worden.

5. DATACOMMUNICATIE

Zoals reeds in de inleiding is vermeld, is er rond de B6700 een omvangrijk datacommunicatienetwerk opgebouwd. Het centrale punt in dit netwerk is de datacommunicatieprocessor (DCP), een kleine special purpose computer met een eigen lokaal geheugen. In het hiernavolgende zal beschreven worden welke software noodzakelijk is voor het besturen van dit netwerk.

5.1. *Network Definition Language*

Burroughs heeft een speciale Algolachtige taal ontwikkeld, de Network Definition Language (NDL). Met behulp van deze taal worden de logische en fysische aspecten van het netwerk beschreven (zoals lijndisciplines, modemtype, transmissie mode, stationtype, ... enzovoorts).
Een aldus verkregen beschrijving van het netwerk wordt aangeboden aan de NDL-compiler die als resultaat twee files aflevert:

- het operating system voor de DCP
- de Network Information File.

De Network Information File wordt regelmatig geraadpleegd en eventueel aangepast door het MCP, zodat deze file altijd de juiste status van het datacommunicatienetwerk aangeeft.

Het datacommunicatiesysteem wordt geactiveerd met een operateursboodschap. Als effect op deze boodschap zoekt het MCP naar de file die het operating system van de DCP bevat. De DCP wordt geactiveerd en vervolgens wordt het operating system van de DCP in het lokale geheugen geladen, waarmee het netwerk operationeel is.

Hoe de communicatie tussen de DCP en het centrale systeem plaatsvindt staat uitgebreid beschreven in DUKE & McINTRY [6] en [7].

5.2. *Message Control Systems*

Het informatieverkeer tussen een station en het centraal systeem wordt bestuurd door Message Control Systems. Welk MCS het informatieverkeer voor een bepaald station bestuurt dient vermeld te worden in de NDL-beschrijving voor dit station.

Enkele functies van een MCS zijn:

- het verzorgen van het inloggen
- het verzamelen van statistieken voor elk station
- het verzorgen van de communicatie tussen een applicatieprogramma en het station
- eventuele text editing faciliteiten verlenen.

De belangrijkste Message Control Systems zijn:

- CANDE voor het interactive gebruik
- THERJE voor de Remote Job Entries
- SATCOM voor de minicomputers
- PLOTMACHINE voor de plotters.

Elk Message Control System is geschreven in DCALGOL. In deze taal (een superset van Burroughs Extended Algol) zijn ten behoeve hiervan als datastructuren MESSAGE en QUEUE opgenomen.
Dit omdat de communicatie tussen een MCS en het MCP plaatsvindt via queues en als informatie-eenheid een message genomen wordt.

6. BEA EN ESPOL

Alle systeemsoftware van de B6700, met uitzondering van de datacommunicatiesoftware, is geschreven in BEA of ESPOL.

6.1. *BEA*

BEA kan in feite beschouwd worden als een superset van Algol 60. De belangrijkste uitbreidingen ten opzichte van Algol 60 zijn:

- type file
- type pointer; met behulp van dit type kunnen karaktermanipulaties uitgevoerd worden
- bitmanipulaties
- macrofaciliteiten
- parallelle processen.
  Synchronisatie kan plaatsvinden met behulp van *events*. Mogelijke acties op events zijn onder andere CAUSE en WAIT. Een uitgebreide beschrijving hierover is te vinden in [1] en [5].
- software interrupts.

Over de uitbreidingen ten opzichte van Algol 60 schrijft BROWN in zijn artikel *Writing Software in Algol* [8]:

"The striking thing about the extensions is that they are generally set at a comparatively low level, heavily oriented towards the B6700 hardware, so that they can be translated into in-line instructions. The level is not so low, however, that it is effectively machine language. A typical character manipulation statement would translate into about ten machine language instructions."

6.2. *ESPOL*

ESPOL (Executive System Problem Oriented Language) is op een paar kleine uitzonderingen na een superset van BEA. In deze taal is onder andere het MCP geschreven. ESPOL biedt dan ook een aantal extra mogelijkheden die in zeer sterke mate gebaseerd zijn op de B6700-hardware, zoals

- directe adressering in het geheugen
- adressering van processorregisters
- lock faciliteiten, uitvoerig beschreven door CLEARY [3]
- het "type" word
- speciale functies die door de IO-processor verwerkt kunnen worden, zoals IO-opdrachten, opdrachten voor het opvragen van de status van de randapparatuur .... enzovoorts.

Beide talen zijn door Burroughs op een redelijke wijze gedocumenteerd [7,8].
Hoewel ik wil aansluiten (ook wat ESPOL betreft) bij de in 6.1. aangehaalde uitspraak van BROWN dient als slotopmerking vermeld te worden, dat het programmeren van systeemsoftware in ESPOL en BEA het leven van de systeem-programmateurs een stuk prettiger heeft gemaakt.

LITERATUUR

[1] ORGANICK, E.I., *Computer System Organization: The B5700/B6700 Series*, Academic Press, New York, 1973.

[2] *Burroughs B6700 Information Processing Systems*, Burroughs Corporation, 1972.

[3] CLEARY, J.G., *Process Handling on Burroughs B6500*, Proceedings of Fourth Australian Computer Conference, Adelaide, South Australia, 1969, 231-239.

[4] HAUCK, E.A. & B.A. DENT, *Burroughs B6500/B7500 stack mechanism*, AFIPS Conference Proceedings 32 (1968) 245-251.

[5] *Burroughs B6700 System Miscellanea*, Burroughs Corporation, 1974.

[6] DUKE, C.S. & M.J. MCINTRY, *A new approach to data communications in a real-time environment*, Proceedings of the IFIP congress 1971, Hardware and Systems section, 84-89.

[7] *Burroughs B6700 Data Communications Functional Description*, Burroughs Corporation, 1970.

[8] BROWN, P.J., *Writing Software in Algol*, Software-Practice and Experience 4 (1974) 139-144.

[9] *Burroughs B6700/B7700 ALGOL Language Reference Manual*, Burroughs Corporation, 1974.

[10] *Burroughs B6700/B7700 Executive System Programming Language (ESPOL) information manual*, Burroughs Corporation, 1972.

MATHEMATICAL CENTRE SYLLABUS 32, 1976, IV, 61-82

VAN BLOKKENDOOS TOT BEDRIJFSSYSTEEM

P. KLINT

1. INLEIDING

Het is nog niet echt lang geleden dat het vertalen en laten executeren van een FORTRAN programma de volgende handelingen vereiste: *)

- plaats de FORTRAN compiler (groene kaarten) in de kaartlezer en druk op de LOAD knop. De FORTRAN compiler wordt ingelezen.
- plaats het te vertalen programma in de kaartlezer. De FORTRAN compiler vertaalt het programma naar een machinekode programma dat op rode kaarten uitgeponst wordt.
- zoek in de kaartenbibliotheek naar een (paars) kaartendeck met het opschrift LOADER en lees dit in.
- plaats het zojuist vertaalde rode object programma in de kaartlezer. De LOADER laadt het in de machine.
- plaats het (blauwe) objectdeck van iedere subroutine die door het programma aangeroepen wordt ook in de kaartlezer. Deze worden ook geladen.
- uiteindelijk start de LOADER de exekutie van het vertaalde FORTRAN programma (waarbij eventueel nog roze datakaarten ingelezen moeten).

Het ongemak van deze methode en het weinig efficient gebruik van apparatuur dat hieruit voortvloeit, heeft ertoe geleid dat allerlei soorten bedrijfssystemen ontworpen en geïmplementeerd zijn, die trachten de computergebruiker zoveel mogelijk diensten te verlenen en apparatuurgebruik te opti-

*) vrij vertaald naar DONOVAN [1] p. 8.

maliseren.

Tegenwoordig bevat een bedrijfssysteem minstens een aantal van de volgende komponenten:

compi-lers	assem-blers	macro-processors	compiler-compilers	text-formatters	zoeken & sorteren
	linkers & loaders	text editor	debug gereedschap	kommando-interpretator	
	file-systeem	scheduler	bibl. systeem aanroepen	accounting backup integriteit	
			drivers ------- computer + randapparaten		

De omgekeerde driehoeksvorm is opzettelijk: alleen de onderste twee lagen behoren tot de kern van het bedrijfssysteem en programma's uit bovenliggende lagen (zoals compilers, editors) worden als gewone gebruikersprogramma's beschouwd. De voordelen van dit schema zijn legio:

- de systeemkern is relatief klein en kan daardoor betrouwbaar zijn, zonder dat hiervoor speciale maatregelen genomen zijn;
- het falen van een editor of compiler verlamt het systeem als geheel niet;
- het is zelfs denkbaar dat gebruikers van een eigen, niet standaard, filesysteem gebruik maken.

In het nu volgende wil ik de eigenlijke bedrijfssysteemkern*) aan een nader onderzoek onderwerpen. De bouwstenen en koncepten die ter sprake komen maken allemaal deel uit van methodes ter konstruktie van logisch konsistente, betrouwbare bedrijfssystemen. Systeemprogrammeermethodologie kortom. Deze methodes zijn niet alleen van belang voor een goede interne werking van een BS, maar beïnvloeden ook de uitwendige vorm. Dit uit zich bijvoorbeeld in:

---

*) afgekort tot BS.

- uniforme programmeerkonventies door het hele systeem; dit betekent dat alle compilers (standaard) kode genereren voor subroutineaanroepen en bijvoorbeeld de naamgeving van files gesystematiseerd is: "file.ftn" voor een file die een fortran programma bevat en "file.obj" voor een file die een objectmoduul bevat.
- het kreëren en synchroniseren van processen is tot op gebruikersniveau mogelijk; hiermee hangt samen dat
  - alle kommando's programmatisch aangeroepen kunnen worden (en de kommandosyntax uniform is!)
  - voor de individuele gebruiker de mogelijkheid bestaat om een eigen kommando-interpretator te maken die wat duidelijker meldingen geeft dan "F302 at 3A4B" *)
  - randapparaten zijn transparant; een programma merkt geen verschil tussen lezen van kaarten, magneetband of disk.

Een aantal van deze onderwerpen is in dit colloquium al aan de orde gekomen of zal nog ter sprake komen. Ik zal mij nu verder beperken tot implementatietechnieken en methodes ter verhoging van betrouwbaarheid en methodes voor meting van systeemprestaties.

2. IMPLEMENTATIEHULPMIDDELEN

2.1 *Systeemprogrammeertaal*

Verbijsterd waren de inwoners van Macondo toen de zigeuner Melquiades voor het eerst staven ijs naar hun dorp bracht [3]. Zo mogelijk nog meer verbazing wekt soms de idee om een BS niet in een assembleertaal te schrijven. Zelfs recente tekstboeken zoals DONOVAN & MADNICK [4] weten zich niet te ontworstelen aan deze historisch gegroeide situatie. Toch zijn er vele voordelen verbonden aan het gebruik van een systeemprogrammeertaal:

- de betrouwbaarheid, onderhoudbaarheid en leesbaarheid van programma's neemt toe;

---

*) BROWN [2] stelt voor om met behulp van een macroprocessor dergelijke kryptische foutmeldingen te vertalen naar meer begrijpelijke meldingen.

- de produktiviteit per programmeur neemt toe; de programmeur houdt een beter overzicht over zijn werk en kan hierdoor betere algoritmen ontwikkelen en implementeren;
- de portabiliteit van programma's neemt toe.

Hiertegen worden wel als voordelen van het gebruik van assembler aangevoerd:

- korte vertaaltijden;
- de kode is erg efficient;
- er kan van speciale hardware eigenschappen gebruik gemaakt worden;
- er bestaat een direkt verband tussen het assemblerprogramma en een eventuele geheugendump.

Naar mijn mening is geen van deze argumenten overtuigend. Metingen in [6] wijzen bijvoorbeeld uit dat bij herschrijven van een bepaald BS van assembler naar SPL *) de SPL versie 1.56 maal groter was geworden. Met meer optimaliserende compilers zal zeker een nog kleinere expansiefaktor bereikt kunnen worden. Bovendien is het niet ongewoon dat 1 percent van een programma verantwoordelijk is voor 50 percent van de exekutietijd, en 10 percent voor 90 percent van de exekutietijd. Na een studie van FORTRAN programma's kwam KNUTH [7] tot de schatting dat de n-de statement in een lijst van statements gesorteerd naar verbruikte exekutietijd verantwoordelijk is voor $(k-1) * k^{-n}$ van de totale exeutietijd, voor zekere k. In zulke gevallen kan herschrijven van een zeer klein aantal subroutines in assembler drastische verbeteringen opleveren.

De toegenomen produktiviteit per programmeur lijkt mij echter van doorslaggevende betekenis. Er spelen hierbij twee, elkaar wederzijds versterkende, faktoren een rol. Aan de ene kant blijkt de produktiviteit per programmeur tamelijk konstant rond 1000-2000 statements per jaar te liggen onafhankelijk van de programmeertaal [8]. Aangezien een statement in een hogere programmeertaal korrespondeert met meerdere statements in assembler, is de feitelijke produktiviteit van de hogere-programmeertaal-programmeur veel hoger dan die van de assembler-programmeur.

---

*) Een bij Philips ontwikkelde, PL/I-achtige systeemprogrammeertaal.

Aan de andere kant is de vereiste hoeveelheid werk voor een bepaald projekt een niet-lineaire funktie van de komplexiteit ervan. Het verband verloopt eerder volgens:

$$\text{werk} = \text{konstante} * (\text{aantal machinekode instrukties})^{1.5}$$

Hieruit blijkt het andere produktiviteitsverhogende aspekt van het gebruik van een hogere programmeertaal: het aantal konceptuele eenheden ("statements") is kleiner, waardoor men langer in het gebied blijft waar de komplexiteitskurve nog een tamelijk lineair gedrag vertoont.

2.2 *Synchronisatieprimitieven*

Twee problemen staan centraal bij de implementatie van een BS, namelijk:

- de synchronisatie van konkurrerende processen, en
- de integriteit van datastrukturen en variabelen.

Deze problemen zijn niet onafhankelijk, want juist het op eenduidige wijze manipuleren met gemeenschappelijke variabelen door konkurrerende processen is van belang.

Er worden voortdurend nieuwe methodes bedacht om synchronisatieproblemen op te lossen. De laatste tijd ligt de nadruk vooral op methodes die het mogelijk maken om synchronisatie en legaal gebruik van gemeenschappelijke variabelen syntaktisch (dwz. met behulp van een compiler) te kontroleren. In de taal CONCURRENT PASCAL van BRINCH HANSEN zijn een aantal van deze methodes gerealiseerd. Om enig idee te geven van dergelijke methodes laten we nu enkele voorstellen van BRINCH HANSEN [9] en HABERMANN [1] de revue passeren. *)

De voorbeelden hebben betrekking op gemeenschappelijke of "shared" variabelen, konditioneel kritische gebieden, event variabelen en pad-expressies en zullen geformuleerd worden in een PASCAL-achtige taal. We presenteren een aantal oplossingen voor steeds hetzelfde probleem: het implementeren van proceskommunikatie door middel van berichtbuffers. De berichten worden bewaard in een cyclische buffer die door een "vulwijzer" en een

*) Het is verstandig om in gedachten te houden dat alle hierna te beschrijven taalprimitieven met behulp van de vertrouwde Dijkstra-semaforen geïmplementeerd kunnen worden.

"leegwijzer" bewaakt wordt. Een zender vult bufferelementen en een ontvanger leegt bufferelementen. Aan de volgende twee voorwaarden moet tenminste voldaan zijn:

- De ontvanger moet geen element legen voordat het gezonden is.
- De zender moet geen element vullen voordat het ontvangen is.

2.2.1 *Gemeenschappelijke variabelen en kritische gebieden*

Een gemeenschappelijke variabele is een variabele waartoe meerder konkurrerende processen toegang hebben. Het is noodzakelijk om per proces exclusief acces tot een gemeenschappelijke variabele te garanderen. Een deklaratie van een dergelijke variabele heeft de vorm:

```
VAR v: SHARED T;
```

waarin "T" een of ander type voorstelt. Operaties op v zijn alleen gedefinieerd binnen een kritisch gebied, dwz. een programmagedeelte waarvan (door de syntaktische struktuur) duidelijk is dat van v gebruik gemaakt wordt. Een kritisch gebied heeft de vorm:

```
REGION v DO s;
```

waarin "s" een willekeurige statement voorstelt. Exclusief acces tot de variabele v wordt gegarandeerd doordat op eenzelfde tijdstip maar één proces zich binnen een kritisch gebied voor variabele v kan bevinden. Bovendien wordt gegarandeerd dat een proces na een eindige tijd een kritisch gebied kan binnen gaan en het ook na een eindige tijd weer verlaat. Merk op dat geen aanname wordt gemaakt omtrent de volgorde waarin processen tot een kritisch gebied toegelaten worden. Ieder redelijk toelatingsbeleid kan gebruikt worden.

Met behulp van bovenstaande hulpmiddelen kunnen nu twee konkurrerende processen die van een gemeenschappelijke apparaat gebruik maken als volgt gesynchroniseerd worden:

```
VAR r SHARED BOOLEAN;
COBEGIN
     "proces p1"
     REPEAT
          REGION r DO gebruik A;
          p1 passief;
     FOREVER
     "proces p2"
     REPEAT
          REGION r DO gebruik A;
          p2 passief;
     FOREVER
COEND
```

2.2.2 *Konditioneel kritische gebieden*

De toegang tot een (enkelvoudig) kritisch gebied wordt verkregen, als men verzekerd is van exclusief acces tot de variabele die met dat gebied geassocieerd is. Een algemener hulpmiddel vormen de konditioneel kritische gebieden, waarmee een proces kan wachten totdat (de komponenten van) een gemeenschappelijke variabele of datastruktuur aan een willekeurige konditie voldoen. Het AWAIT taalelement kan gebruikt worden om konditioneel kritische gebieden te implementeren. De syntaktische vorm is:

```
VAR v: SHARED T;
REGION v DO
BEGIN ... AWAIT b; ... END
```

Een await is alleen binnen een kritisch gebied toegestaan. Men spreekt van een konditioneel kritisch gebied als het eerste exekuteerbare statement van een kritisch gebied een AWAIT bevat. Het berichtbuffer-probleem kan nu als volgt met konditioneel kritische gebieden opgelost worden:

```
TYPE B = SHARED RECORD
     buffer: ARRAY 0..N-1 OF T;
     p,c: 0..N-1;
     full: 0..N-1;
END

PROCEDURE send(m:T; VAR b: B);
REGION b DO
     BEGIN
          AWAIT full < N;
          buffer (p) := m;
          p := (p+1) MOD N;
          full := full + 1;
     END

PROCEDURE receive (VAR m: T; b: B)
REGION b DO
     BEGIN
          AWAIT full > 0;
          m := buffer (c);
          c := (c+1) MOD N;
          full := full - 1;
     END
```

2.2.3 *Event variabelen*

Hoewel de bovenstaande oplossing elegant en kompakt is heeft hij een belangrijk nadeel: men heeft geen kontrole over de manier waarop processen die een AWAIT uitgevoerd hebben, aan de beurt komen zodra aan hun wachtkonditie voldaan is. Om de scheduling van wachtende processen expliciet in handen te krijgen moet de programmeur kunnen beschikken over een rij gebeurtenissen of events die geassocieerd zijn met een gemeenschappelijke variabele. Zo'n rij wordt gedeklareerd door:

```
VAR e: EVENT v;
```

Een proces kan een kritisch gebied, geassocieerd met variabele v, verlaten en zichzelf in de wachtrij e plaatsen door de standaardroutine

```
sleep (e)
```

aan te roepen. Een proces kan alle processen in de wachtrij voor e toestaan hun kritische gebied weer te betreden door de standaardroutine

```
wakeup (e)
```

uit te voeren. Sleep en wakeup mogen alleen aangeroepen worden in een kritisch gebied geassocieerd met de, in de deklaratie van het event e genoemde, gemeenschappelijke variabele v.

De oplossing van het berichtbuffer-probleem kan nu als volgt met behulp van event variabelen geformuleerd worden:

```
TYPE B = SHARED RECORD
     buffer: ARRAY 0..N-1 OF T
     p,c: 0..N-1;
     VAR full, empty: EVENT B;
END

PROCEDURE send(m: T; VAR b: B);
REGION b DO
     BEGIN
          IF (full = N) THEN sleep (full);
          buffer (p) := m;
          p := (p+1) MOD N;
          full := full + 1;
          IF (full = 1) THEN wakeup (empty);
     END

PROCEDURE receive (m: T; VAR b: B);
REGION b DO
     BEGIN
          IF (full = 0) THEN  sleep (empty);
          m := buffer (c);
          c := (c+1) MOD N;
          full := full - 1;
          IF (full = N - 1) THEN wakeup (full);
     END
```

2.2.4 *Datastrukturen*

Het gebruik van event variabelen leidt tot een overzichtelijk en efficiente oplossing van synchronisatieproblemen, waarin de scheduling expliciet is. Toch kunnen tegen deze oplossing twee bezwaren aangevoerd worden; zeker als men deze oplossing bekijkt vanuit het momenteel populaire gezichtspunt van verdeel-en-heers-programmeren:

- de manipulatie van vul- en leegwijzer heeft uitsluitend betekenis voor de ringbuffer; definitie van deze operaties behoort eigenlijk deel uit te maken van de ringbufferdefinitie;
- het afdwingen van de volgorde schrijven-lezen van een bufferelement zou ook tot de ringbufferdefinitie moeten behoren.

Voor beide bezwaren zijn oplossingen bedacht. Met het "class" koncept uit SIMULA kan het eerste punt ondervangen worden. Een class definitie bestaat uit de definitie van een gestruktureerd datatype tezamen met alle zinvolle operaties hierop. Een class definitie heeft de vorm:

```
CLASS T = v1: T1; v2: T2; ... ; vm: Tm;
PROCEDURE p1(...) BEGIN s1 END
  ...
PROCEDURE pn(...) BEGIN sn END
BEGIN s0 END
```

en definieert een datastruktuur van type T bestaande uit de komponenten $v_1,\ldots,v_m$ van type $T_1,\ldots,T_m$ en een verzameling procedures $p_1,\ldots,p_n$ die op de datastruktuur kunnen opereren. Op het moment van deklaratie van een variabele van type T:

```
VAR v: T;
```

wordt ruimte geallokeerd voor de komponenten $v_1,\ldots,v_m$ en wordt de initialisatie statement s_0 uitgevoerd. Een aanroep van de procedure p_i "binnen" de variabele v wordt genoteerd als

```
v.pi(...)
```

Een uitgebreider voorbeeld van een class definitie wordt in de volgende paragraaf behandeld.

2.2.5 *Pad-expressies*

Pad-expressies [11] dienen als hulpmiddel voor het afdwingen van een volgorde waarin bepaalde operaties op een datastruktuur uitgevoerd moeten worden. De idee die aan pad-expressies ten grondslag ligt is als volgt te beschrijven: associeer met iedere class een regelmechanisme plus beschrijving van de volgorde waarin de procedures, die binnen de class gedefinieerd zijn, aangeroepen mogen worden. Probeert een proces een procedure van een class uit te voeren dan kontroleert het regelmechanisme of aan de volgordebeschrijving voldaan is; zoniet dan wordt het proces vertraagd tot hieraan wel voldaan is.

De volgordebeschrijving wordt kortweg pad-expressie genoemd en is opgebouwd uit:

";" om sequentiele akties te beschrijven,

"," om selektie uit een verzameling akties te beschrijven,

"(" en ")" voor groepering.

De pad-expressie:

```
PATH p; (q,r); s END
```

synchroniseert de exekuties van de procedures p, q, r en s, en laat slechts twee exekutievolgordes toe:

p gevolgd door q gevolgd door s

of

p gevolgd door r gevolgd door s.

Toepassing van class koncept en pad-expressies levert de laatste oplossing voor het ringbufferprobleem:

```
CLASS buffer =
     VAR frame: M;
     PATH write; read END;

     PROCEDURE read (VAR m: M);
          BEGIN m := frame END;

     PROCEDURE write(m: M);
          BEGIN frame := m END;

END

CLASS ringbuffer =
     r:   ARRAY 0..N-1 OF buffer;

     CLASS index =
          VAR p:    INTEGER;
          PATH next END;

          FUNCTION next: INTEGER;
               BEGIN p := (p+1) MOD N; next := p END;

               BEGIN p := 0 END
          END

          VAR writeslot, readslot: index;

     PROCEDURE send (m: M);
          BEGIN r [writeslot.next].write (m) END;

     PROCEDURE receive (VAR m: M);
          BEGIN r [readslot.next].read (m) END;
END
```

3. BETROUWBAARHEID

3.1 *Betrouwbaarheid versus korrektheid*

Het is, voorzichtig gezegd, een gemeenplaats om op te merken dat de betrouwbaarheid van een BS van het grootste belang is. Toch leert de ervaring dat het aan deze betrouwbaarheid nog wel eens schort.

Er is een groot verschil tussen een korrekt programma en een betrouwbaar programma. Een programma is korrekt als het aan zijn specificaties voldoet; maar deze specifikaties kunnen onvolledig of niet korrekt zijn. Een programma is betrouwbaar als het waarschijnlijk is dat, ook onder niet voorziene bedrijfsomstandigheden het zijn taken goed vervult of tenminste zijn

onvermogen om een taak uit te voeren meldt. Betrouwbaarheid is dus een statistische grootheid. Kort samengevat doet een korrekt programma wat de programmeur bedoelt en een betrouwbaar programma wat de gebruiker bedoelt.

Het is momenteel, praktisch gesproken, onmogelijk om een korrektheidsbewijs te leveren van een BS en bovendien zou een dergelijk bewijs niet noodzakelijk de betrouwbaarheid van dat systeem impliceren. Dit komt aan de ene kant doordat de koncepten die bijvoorbeeld in een BS gebruikt worden (virtueel geheugen, i/o apparaten, filesysteem) zo ingewikkeld zijn dat formalisering van hun semantiek haast uitgesloten lijkt. Aan de andere kant geeft een korrektheidsbewijs, zeker zolang de bewijsmethode zelf niet gemechaniseerd is, geen garantie tegen triviale pons- en spelfouten.

3.2 *Betrouwbaarheid van hardware*

Oorzaken van systeemstoringen zijn: hardware fouten, software fouten of een kombinatie hiervan. In sommige gevallen is het niet eens a priori duidelijk in welke kategorie een storing valt.

Er zijn vele methodes voor opsporen en vermijden van hardware storingen, zoals uit enkele voorbeelden blijkt:

- toepassing van foutdetekterende en- herstellende kodes, zowel in geheugen, randapparaten als in de CPU zelf;
- operaties door meerdere subsystemen uit laten voeren en een scheidsrechter, op grond van deze uitkomsten, laten beslissen wat het resultaat van deze instruktie moet zijn (zie 3.3);
- dynamische herkonfiguratie bij storingen. Een defekt geheugenmoduul of een defekte processor worden, dynamisch, buiten werking gesteld. De Burroughs B7600 maakt van deze methode gebruik.

Het is onvermijdelijk dat toegenomen komplexiteit van niet-redundante hardware verminderde betrouwbaarheid impliceert. Vergelijk een televisietoestel met een computer. Niemand zal het als onoverkomelijk ervaren dat een televisietoestel eenmaal per vijf jaar defekt is. Als we aannemen dat een computer 10.000 maal zoveel komponenten bevat, dan zou deze eenmaal per vier uur defekt mogen zijn! Ik probeer hiermee alleen maar aan te tonen dat de eisen die we aan de hardware stellen erg hoog zijn (en de eisen die we aan de software stellen veel te laag?).

3.3 *Redundantie*

We zullen nu aantonen hoe betrouwbaarheid te bereiken is door middel van redundantie. De redenering is afkomstig van VON NEUMANN [12] en heeft betrekking op logische netwerken; toepassing op programmatuur is echter zeker denkbaar.

Een eenvoudige vorm van redundantie is drievoudige redundantie (TMR: Triple Module Redundancy). Een schakeling S wordt vervangen door drie kopieën S_1, S_2 en S_3 tezamen met een meerderheidskiezer MK. De meerderheid van de uitkomsten van $S_1,\ldots,S_3$ wordt als uitkomst van de gehele schakeling beschouwd. De meerderheidskiezer is een universeel schakelelement (dit betekent: voor iedere logische schakeling bestaat een equivalente schakeling die alleen uit meerderheidskiezers opgebouwd is) en daarom is het optreden van fouten in een MK representatief voor het gedrag van fouten in een logische schakeling in het algemeen.*)

Laten N_1, N_2 en N_3 $(0<N_i\leq 1)$ een bovengrens vormen voor de waarschijnlijkheid dat ingang i van de MK een verkeerd signaal draagt en laat E de waarschijnlijkheid aanduiden voor het falen van de MK zelf. Dan is $E+N_1+N_2+N_3$ de meest algemene bovengrens vormen voor de kans op fouten als signalen zich voortplanten door een enkele MK in een schakeling. Onder de speciale aanname dat

(1) de waarschijnlijkheden van fouten in de ingangen onafhankelijk zijn, en

(2) alle ingangslijnen hetzelfde signaal dragen

is

$$T = N_1 \times N_2 + N_1 \times N_2 + N_2 \times N_3 - 2 \times N_1 \times N_2 \times N_3$$

een bovengrens voor de kans dat minstens twee ingangen een verkeerd signaal dragen en dus is

$$E = (1-E)\times T + E\times(1-T) = E + (1-2E)\times T$$

---

*) De nu volgende beschouwing heeft betrekking op zogenaamde McCULLOCH-PITTS neuronen [13], die stimulerende en verbiedende ingangen kennen. Hierdoor is negatie a priori als primitieve funktie beschikbaar en hoeft niet apart gekonstrueerd te worden.

een kleinere bovengrens voor het falen van de uitgangslijn. Als nu tenslotte alle $N_i<N$ dan is E+3N de algemene bovengrens en

$$E + (1-2E) \times (3N^2-2N^3) \leq E + 3N^2$$

de bovengrens voor het speciale geval.

De implikaties van het bovenstaande zijn duidelijk. In het algemene geval neemt de kans op fouten toe naarmate de schakelingsdiepte toeneemt omdat E+3N>N. In het speciale geval is de situatie $E+3N^2<N$ mogelijk (en noodzakelijk) om te voorkomen dat het systeem degenereert tot totale irrelevantie. Het blijkt dat een uiteindelijk foutenniveau van 2% basiskomponenten met een foutenniveau van 0.4% vereist.

Bovenstaande methode is onpraktisch, want iedere basiskomponent wordt verdrievoudigd. Een schakeling met M komponenten moet op deze manier 3^M komponenten bevatten. Een faktor 3M is te bereiken in "gemultiplexte" schakelingen. Hierin wordt iedere signaallijn in de oorspronkelijke schakeling vervangen door een bungel van meer lijnen. Bovendien wordt het begrip "lijn draagt signaal" vervangen door "V% van de lijnen in de bundel dragen een signaal". Voor V = 93% geeft dit:

lijnen/bundel	foutkans
1000	2.7×10^{-2}
5000	4×10^{-3}
10000	1.6×10^{-10}
20000	2.8×10^{-19}

Hieruit blijkt dat een willekeurige betrouwbaarheid te bereiken is. Toepassing met bundels met 20.000 lijnen wordt wellicht haalbaar naarmate de ontwikkelingen in de microminiaturisering voortschrijden. Bovendien is Von Neumann's methode niet maximaal efficient en zal met efficientere methodes de bundelgrootte nog verkleind kunnen worden.

3.4 *Betrouwbaarheid van software*

De reaktie van het BS op hardware storingen hangt nauw met de aard van de storing samen. Een falend disk- of tapetransport kan enkele malen herstart worden om te zien of het transport een volgende keer wel slaagt. Random interrupts van een randapparaat kunnen geregistreerd en weggegooid

worden, maar bij storingen in het apparaat waar het systeem zelf resideert of bij stroomstoringen kan hooguit een "elegante degradatie" bereikt worden. Dit wil zeggen dat de toestand van het systeem bevroren wordt op dat bij herstarten de oude toestand zoveel mogelijk hersteld kan worden.

Software fouten moeten zo snel mogelijk ontdekt en hersteld worden. In het algemeen geldt dat door een vroegtijdige detektie van fouten verbreiding ervan door het hele systeem vermeden kan worden.

Ook hier is een of andere vorm van redundantie nodig om fouten snel op te sporen. Redundantie in tabellen en datastrukturen is soms op eenvoudige wijze te realiseren. Redundantie in programma's is moeilijker te bereiken: het is duidelijk dat de betrouwbaarheid van een programma niet vergroot wordt door meerdere kopieën van dat programma te gebruiken, vooral omdat softwarestoringen altijd ontwerp- dan wel implementatiefouten zijn. Een radio werkt niet beter met twee defekte transistors inplaats van een. Beschrijving van de effekten, die het programma tot stand moet brengen, in termen van verschillende implementatiemethodes draagt daarentegen wel bij tot een redundantie en foutwerende struktuur. Bovendien draagt redundantie in de programmeertaal ook bij tot het vermijden van fouten. De vergissing

```
FOUT = FOT + 1
```

blijft in FORTRAN (tijdens compilatie) onopgemerkt, maar wordt wel ontdekt in een programmeertaal met verplichte deklaratie van variabelen. Een plus-and-becomes operator sluit deze vergissing ook uit. Een (niet bedoelde) diskrepantie tussen array grenzen en grenzen in een forstatement, zoals

```
INTEGER ARRAY a[0:10];
FOR i := 1 STEP 1 UNTIL 10 DO a[i] := 13;
```

is in ALGOL mogelijk. In een taal met de konstruktie

```
FOR ALL INDICES i DO a[i] := 13;
```

is het minder gemakkelijk deze fout te maken. Deze voorbeelden maken duidelijk dat de ontwikkeling van programmeertalen een belangrijke bijdrage kan leveren tot het konstrueren van foutenarme programma's.

3.4.1 *Ad hoc methodes*

In de loop der tijden is een aantal "truukjes" ontwikkeld om (lokaal) een zekere mate van redundantie in programma en data te bereiken. De volgende voorbeelden geven een indruk van de mogelijkheden die er zijn:

- positieve tests: op plaatsen waar een programma zich vertakt op grond van de waarde van een variabele (zoals in een case statement) wordt alleen een pad gekozen als resultaat van een positieve test; als geen van de tests van toepassing is wordt een foutmelding gegeven.
- range kontrole: op grond van hun semantische betekenis kunnen sommige variabelen maar een bepaald aantal waarden aannemen; pointers mogen alleen binnen een bepaald geheugentrajekt wijzen.
- konsistentiekontrole: periodieke kontrole van alle datastructuren. Als de jobscheduler van een verminkte administratie gebruik maakt, krijgen sommige jobs een zeer goede en andere helemaal geen behandeling.
- checksum op tabellen: pas de checksum aan bij iedere tabelwijziging en bereken regelmatig een nieuwe checksum voor de gehele tabel en vergelijk deze met de oude waarde. Op deze manier kunnen illegale tabelmanipulaties opgespoord worden.
- linkverdubbeling: gebruik, waar een enkelvoudig gelinkte lijst zou volstaan, een dubbelgelinkte lijststruktuur met terugverwijzingen, zodat de lijstsruktuur eventueel weer hersteld kan worden; bovendien maakt de extra terugverwijzing kontrole van de integriteit van de lijst eenvoudig.
- pointersleutels: associeer met pointers die naar objekten van een bepaald type wijzen en met die objekten zelf een vast bitpatroon (en meer in het algemeen het gebruik van tagbits).
- geheugenprotektie: scheiding van data en instrukties; instrukties execute only.

Naast dergelijke ad hoc methodes wordt momenteel geëxperimenteerd met programmeersystemen waarin redundante informatie en foutdetektie geïnte-

greerd zijn.

3.4.2 *Recoveryblokken*

We zullen nu een voorbeeld van een dergelijk programmeersysteem bespreken [14, 15]. Het centrale concept in dit systeem is het "recoveryblok", dat dezelfde rol speelt als het ALGOL blok, maar een meer komplexe interne struktuur heeft ten behoeve van foutdetektie en- herstel. Een recoveryblok bestaat uit:

- een primair blok, dat korrespondeert met het ALGOL blok, en de vereiste operaties uitvoert.
- een akseptatietest, die uitgevoerd wordt na afloop van exekutie van het primaire blok, om te kontroleren of het primaire blok zijn taken op bevredigende wijze uitgevoerd heeft. De akseptatietest is geen assertie, in die zin dat het een praktische test is en geen 100% zekerheid biedt. De test zelf kan ook fout zijn!
- een of meerdere alternatieve blokken, die geëxekuteerd worden als de akseptatietest faalt. Na exekutie van een alternatief blok wordt de akseptatietest herhaald.

In deze benadering is alleen het effekt van primaire of alternatieve blokken op de omgeving (wijziging van globale variabelen) relevant en blijft de implementatiewijze per blok buiten beschouwing. De akseptatietest heeft dan ook geen toegang tot lokale variabelen van een blok. Een primair of alternatief blok kan ook weer (geneste) recoveryblokken bevatten. Een recoveryblok als geheel faalt als de akseptatietest heeft gefaald na exekutie van primair blok en alle alternatieve blokken.

Bij implementatie van deze methode doen zich drie problemen voor:

- de extra kode die door akseptatietest en alternatieve blokken geïntroduceerd wordt, mag de omvang van het systeem als geheel niet buiten verhouding doen toenemen;
- bij falen van een blok moet de oorspronkelijke globale omgeving voor exekutie van dit blok hersteld worden;
- het bewaren van de exekutieomgeving moet efficient zijn, anders is de methode niet praktisch toepasbaar.

Het eerste punt dikteert het gebruik van een of ander virtueel geheugensysteem, waarin de niet aktieve alternatieve blokken zich op achtergrondgeheugen bevinden. Bovendien moet de i/o "gespoold" zijn omdat anders i/o operaties niet meer ongedaan gemaakt kunnen worden.

Beide andere punten illustreren dat het maken van een herstartbare geheugendump voor ingang van een recoveryblok, of achterwaartse exekutie na het falen van een recoveryblok niet in aanmerking komen.

Om dit probleem op te lossen kan men gebruik maken van een techniek die toegepast wordt in zogenaamde "cache" geheugens; de cyclustijd van het (kern)geheugen wordt hiermee verkleind door bij ieder geheugenacces eerst te kijken of de inhoud van het betreffende adres misschien al in het veel snellere cache geheugen aanwezig is. Op analoge wijze kan de omgeving voor exekutie van een recoveryblok bewaard worden. Associeer met ieder (virtueel) geheugenadres een Boolean die aangeeft of de inhoud van dat adres al in de cache aanwezig is. Bij ingang van een recoveryblok worden alle Booleans op FALSE gezet; iedere schrijfoperatie test de Boolean en zet, als deze FALSE is, virtueel adres en huidige waarde in de cache. Als een blok faalt bevat de cache juist voldoende informatie om de oude waarden van gewijzigde globale variabelen te herstellen. Omdat recoveryblokken genest kunnen voorkomen moet de cache stapelsgewijs georganiseerd worden.

Het is belangrijk op te merken dat het uitlezen van geheugenlokaties geen extra tijd vergt en het assigneren aan lokale variabelen evenmin duurder wordt. Alleen toekenningen aan globale variabelen en blokingang en-uitgang kosten meer tijd. Door de eenvoud van de methode komt deze zeker ook voor hardware implementatie in aanmerking, waardoor deze nadelen minder zwaar wegen.

4. METINGEN

Naast methodes die dienen voor het opsporen of vermijden van logische fouten en kodeerfouten in de implementatiefase van een BS, is het belangrijk te beschikken over hulpmiddelen die tijdens het gebruik van een BS fouten opsporen en prestaties meten.

Het verrichten van metingen is om twee redenen van belang. In de eerste plaats zijn in de ontwerpfase beslissingen genomen, gebaseerd op theoretische verwachtingen over het dynamische gedrag van het systeem. Als metingen nu een ander gedrag te zien geven, kan dit leiden tot herziening van de oor-

spronkelijke ontwerpbeslissing of tot verbeterde implementatie van bepaalde systeemmodulen. Een typisch voorbeeld is het uitvoeren van optimalisaties die zo tijdrovend of ingewikkeld zijn dat door de optimalisatie beoogde besparingen weer teniet worden gedaan. Stel dat een sterk optimaliserende diskdriver rekening houdt met de rotationele positie van de disk ten opzichte van de leeskop en diskverzoeken op grond hiervan sorteert. Als dit sorteerproces teveel tijd kost kan, wanneer uitgerekend is welke sektor gelezen moet worden, de leeskop de betreffende sektor net gepasseerd zijn en moet zeker een hele omwenteling gewacht worden!

In de tweede plaats is het noodzakelijk metingen van het dynamische gedrag van een BS te kunnen doen, uitgaande van een bepaalde hardwarekonfiguratie of bepaalde werkomstandigheden. Dit maakt het mogelijk een systeem zo goed mogelijk op lokale eisen af te stemmen.

Zoals een ampèremeter, door zijn inwendige weerstand, de elektrische eigenschappen van een schakeling beïnvloedt, zo wordt het gedrag van een BS meestal beïnvloed door toe te passen meetmethodes. De beste, maar vaak moeilijk te realiseren, methode is het gebruik van een tweede computer die het BS in de eerste machine bespionneert. Tot de mogelijkheden behoren:

- met vaste tijdsintervallen de programcounter of de naam van de op dat moment geëxecuteerde routine registreren;
- symbolische inspektie van systeemtabellen en datastrukturen;
- voortdurende konsistentiekontrole van het gehele systeem (zonder dat het systeem hierdoor extra belast wordt);
- detektie van gebeurtenissen, zonder dat hiervoor in het systeem voorzieningen zijn aangebracht (meting van hoeveelheid swapping, i/o tijd).

Helaas is deze methode vaak niet te verwezenlijken en moet men zijn toevlucht nemen tot primitievere middelen. Enkele van deze middelen komen nu ter sprake.

Simulatie van een BS is gebaseerd op een model van dit systeem en meestal een aantal hypothesen die dienen om het model te vereenvoudigen en hanteerbaar te maken. Op grond van het (vereenvoudigde) model worden voorspellingen gedaan over het dynamsiche gedrag van het systeem in kwestie. Hoewel bruikbare resultaten bereikt kunnen worden, heeft deze methode een aantal in het oog springende nadelen. Aan de ene kant is konstruktie van

model, eventuele vereenvoudigingen en simulatieprogramma verre van triviaal. Aan de andere kant is er geen zekerheid dat het model met de werkelijkheid overeenstemt. Dit telt met name in de ontwerpfase zwaar als de specifikaties van het systeem nog aan wijzigingen onderhevig zijn.

De twee nog te bespreken methodes kunnen interne en externe meetmethode genoemd worden. De interne meetmethode eist dat alle systeemmodulen een zekere hoeveelheid statistiek bijhouden, wanneer ze aangeroepen worden. Dit betekent dat de metingen door het systeem zelf verricht worden, met als voordeel dat alleen relevante (lees: op het moment van ontwerp relevant geachte) gegevens geregistreerd worden en bij analyse van de gegevens geen datareduktie-technieken toegepast hoeven worden. Nadelen zijn: starheid, mogelijke programmeerfouten tijdens het uitvoeren van de statistiek en de extra kode die nodig is om de statistiek bij te houden.

De externe meetmethode bestaat uit vertalen van systeemmodulen onder speciale compileropties. Denk hierbij aan het genereren van kode die bij routine in- en uitgang een klokroutine aanroept om exekutieprofielen te verkrijgen of aan kode die wijziging van bepaalde variabelen of tabellen registreert.*) Deze methode is flexibeler dan de vorige; het is ook mogelijk de metingen tot een enkel moduul te beperken. Het bezwaar van de extra kode blijft gelden maar kan toch enigszins omzeild worden.

5. SLOTOPMERKING

In het voorafgaande zijn een aantal hulpmiddelen ter sprake gekomen, die nuttig zijn bij ontwerp of implementatie van een BS. Naar volledigheid is uiteraard niet gestreefd. Afgezien van deze zuiver technische aspekten zijn er ook andere faktoren die succes of mislukking van een BS bepalen.

Op het gebied van organisatie en dokumentatie van een groot systeemprogrammeerprojekt moet nog veel geleerd worden. Een interessante ontwikkeling vormen zogenaamde "zelfdokumenterende" systemen; hiervoor worden programmeertalen gebruikt waarin ook aan de vorm van kommentaar syntaktische eisen worden opgelegd. Deze kommentaarsyntax dwingt zowel tot het schrijven van kommentaar (door een zekere kommentaardichtheid of kommentaar bij procedureverklaraties te eisen) als tot strukturering ervan. Op deze manier kunnen automatisch manuals gegenereerd worden voor gebruikers van verschillende kennisniveaus.

---

*) Alweer een voordeel van het gebruik van een hogere programmeertaal!

LITERATUUR

[1] DONOVAN, J.J., *Systems programming*, McGraw-Hill, New York, 1972.

[2] BROWN, P.J., *Macroprocessors and techniques for portable software*, John Wiley & Sons, London, 1974.

[3] GARCIA MARQUEZ, G., *Honderd jaar eenzaamheid*, Meulenhof, 1972.

[4] MADNICK, S.E. & J.J. DONOVAN, *Operating Systems*, Mcgraw-Hill, New York, 1974.

[5] VAN DER POEL, W.L. & J.J. MAARSEN, *Machine oriented higher level languages*, *North-Holland*, Amsterdam, 1974.

[6] KLUNDER, J., *Experiences with SPL*, in [5].

[7] KNUTH, D.E., *An empirical study of FORTRAN programs*, Software - Practice and Experience 1 (1971) 1, 105-133.

[8] BROOKS JR, F.P., *The mythical man-month*, DATAMATION 20 (1974) 12, 44-52.

[9] BRINCH HANSEN, P., *Operating system principles*, Prentice-Hall, New Jersey, 1973.

[10] GELENBE, E. & C. KAISER, (ed.), *Operating systems*, proceedings of an international symposium held at Rocquencourt, april 23-25, 1974, Springer-Verlag, Berlin, 1974.

[11] CAMPBELL, R.H. & A.N. HABERMANN, *The specification of process synchronisation by path expressions*, in [10].

[12] VON NEUMANN, J., *Probabilistic logics and the synthesis of reliable organisms from unreliable components*, in Automata studies, Princeton University Press, 1956 (herdrukt in A.H. Taub (ed.), *John von Neumann collected works*, vol. 5, Pergamon Press, 1963).

[13] MINSKY, M., *Computation: finite and infinite machines*, Prentice-Hall, 1972.

[14] HORNING, J.J. et al., *A program structure for error detection and recovery*, in [10].

[15] RANDELL, B., *System structure for fault tolerant software*, in Proceedings international symposium on reliable software 1975, SIGPLAN 10 (1975) 6, 437-449.

MATHEMATICAL CENTRE SYLLABUS 32, 1976, V, 83-100

MULTIPROCESSING IN DE CYBER COMPUTERS

H. BARREVELD
Stichting Academisch Rekencentrum Amsterdam

1. INLEIDING

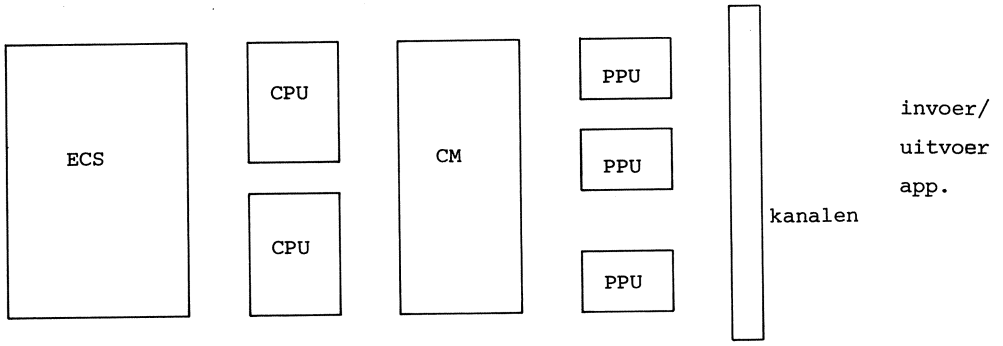
Deze uiteenzetting gaat over het "binnenwerk" van het SCOPE besturingssysteem voor de CDC Cyber computers. Deze computer en dit besrijfssysteem dateren in aanleg uit de eerste helft van de zestiger jaren. In die tijd, toen men bijvoorbeeld nog nooit gehoord had van iets als 'cooperating sequential processes' was deze machine, waarin een groot aantal processoren gezamenlijk de besturingsfunkties verricht, een bijzonderheid. Deze opzet, die ook in andere opzichten dan het aantal processoren naar voren komt, is destijds gekozen om met naar huidige begrippen tamelijk langzame electronische componenten toch een hoge effectieve verwerkingssnelheid te kunnen bereiken.

In wat hierna volgt zal worden uiteengezet op welke wijze de taken met elkaar communiceren die in verband met de besturingsfunktie worden uitgevoerd. Hiervoor is het nodig eerst een globale uiteenzetting te geven van de werking van de hardware. Ook hierbij zal reeds een aantal vormen van commumicatie tussen gelijktijdig uitgevoerde processen naar voren komen; bijvoorbeeld tussen een randapparaat en een 'peripheral processor'.

2. DE COMPUTER

Met weglating van een aantal voor deze uiteenzetting niet relevante onderdelen is de computer opgebouwd als in de figuur bovenaan de volgende bladzijde. Dat wil zeggen:

- Een of twee centrale verwerkingseenheden (CPU's). Ze worden gebruikt voor de verwerking van gewone programma's, bijvoorbeeld alle gebruikersprogramma's. Ze maken hierbij gebruik van het centrale geheugen (CM), dat maximaal 128k woorden van 60 bits omvat. Het achtergrondgeheugen (ECS) wordt

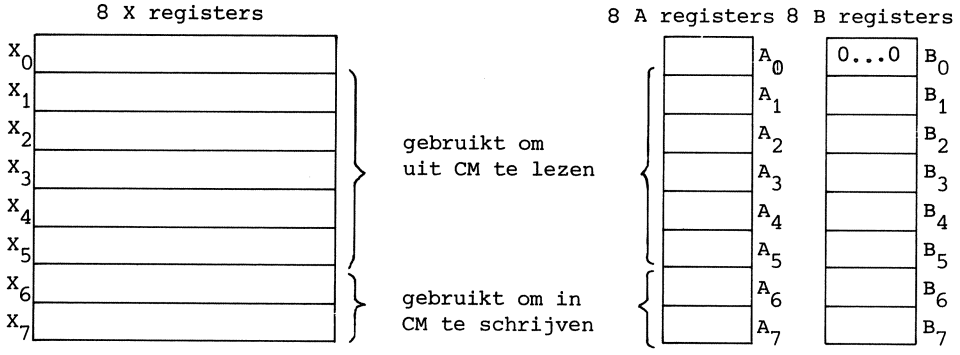


gebruikt voor buffers, het bewaren van tijdelijk niet uitgevoerde programma's, etc.

- Tien, veertien, zeventien of twintig perifere verwerkingseenheden (PPU's). Deze verzorgen de besturingsfunkties en de in- en uitvoer van en naar randapparaten, zoals magneetbandeenheden, schijfgeheugens, terminals, de operator console. Ze hebben elk een 4k geheugen van 12-bits-woorden, en kunnen ook overal in CM lezen en schrijven.

2.1. De computer - centrale verwerkingseenheid

De centrale processors beschikken elk over de volgende registers:



- Acht A (adres) registers, die gebruikt worden voor adressering van het geheugen. Ze hebben een lengte van 18 bits.
- Acht X (operand) registers, die voor arithmetische bewerkingen gebruikt

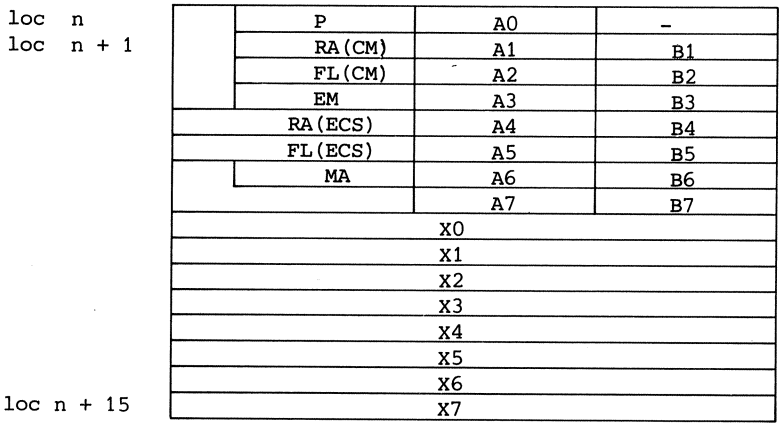
worden. De lengte is 60 bits, hetgeen overeenkomt met de woordlengte van het centrale geheugen.

- B (index) registers, ook 18 bits lang.

Eenvoudige arithmetiek, optellen en aftrekken, is mogelijk met de A, B en X registers. Hiermee kunnen eenvoudig adresberekeningen worden uitgevoerd, die bij het laden van een adres in een A register leiden tot een geheugenacces van of naar het bij dat A register behorende X register. Alle arithmetiek met "echte", 60-bits-operanden vindt uitsluitend van en naar X registers plaats. De verzameling arithmetische instructies voor 60-bits-operanden vertoont een aantal voor moderne computers wellicht opmerkelijke lacunes, die duidelijk maken dat de computer is ontworpen voor wetenschappelijk rekenwerk. Zo zijn er geen instructies voor manipulaties aan tekenrijen (de 'compare-move unit' die voor dit doel bestaat werd eerst vrij kort geleden toegevoegd), en ontbreken instructies voor integer vermenigvuldigen en delen met dubbellengte-produkt en deeltal. Tengevolge van de grote woordlengte is de nauwkeurigheid van de floating point arithmetiek echter groot: 48 bits mantisse, 12 bits exponent (dubbellengte-arithmetiek zelfs 96 bits mantisse!).

Uit wat hiervoor gezegd is over de globale architectuur zal het te begrijpen zijn dat de centrale processor geen I/O instructies kent. De centrale processor heeft ook geen (hardware) middelen om informatie aan een PPU of aan de andere centrale processor te geven. Het enige dat hij kan doen is op een afgesproken plaats iets in het geheugen zetten, en afwachten wanneer de ander het "hebben" wil. Dit hangt samen met de filosofie die aan het gebruik van de hardware ten grondslag ligt: de PPU's besturen de machine, de CPU's zijn domme rekenkolossen, die doen wat hun wordt opgedragen. De PPU's kunnen dan ook wél een CPU een bepaalde taak opleggen. Dit gebeurt door de CPU een zg. exchange jump te doen uitvoeren. Hierbij wordt de volledige status van de CPU, dat wil zeggen de inhoud van de A, X en B registers, de opdrachtenteller en nog een aantal andere registers verwisseld met de inhoud van een blok van 16 geheugenwoorden. De correspondentie tussen de registers en de 16 geheugenwoorden, het 'exchange package', is als in de figuur bovenaan de volgende bladzijde. Twee nog onbekende paren registers springen in het oog: de beide RA en FL registers, waarvan één paar kennelijk betrekking heeft op het centrale geheugen en het andere paar op het achtergrond-kernengeheugen. Deze registers hebben te maken met het geheugenbeheer van de computer. De machine is bedoeld om met een multiprogrammeringssysteem te werken. Daarin wordt af-

wisselend aan een aantal taken tegelijk gewerkt; een aantal hiervan bevindt zich tegelijkertijd in het centrale geheugen. Bij deze computer krijgt iedere taak (of job) een fysiek aaneengesloten stuk geheugen toegewezen, waarvan de lengte tijdens de verwerking van een taak al naar behoefte groter of kleiner kan worden. Eventuele konflikten die hierbij ontstaan doordat een

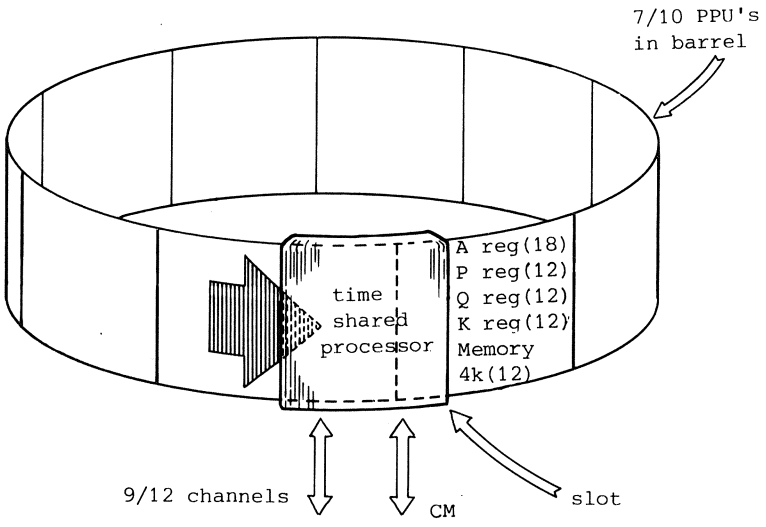


aantal taken opeenvolgende stukken geheugen bezet, worden opgelost door de stukken geheugen in hun geheel te verplaatsen. Voor de processor die aan zo'n taak rekent blijft dit echter onzichtbaar, dank zij de RA en FL registers. Het besturingssysteem zorgt er namelijk voor dat RA het fysieke beginadres van zo'n stuk geheugen bevat en FL de lengte (Reference Address en Field Length). Alle geheugenaccessen die gedaan worden, worden door de hardware met de inhoud van RA geïndiceerd, zodat voor de processor de adressen van het stuk geheugen lopen van 0 tot FL, ongeacht de fysieke plaats van dit stuk in het geheugen.
Een nadeel van deze techniek is dat het verschuiven van de stukken geheugen processortijd kost; voordelen zijn het vermogen tot aanpassing aan de behoefte van een job en een volledige bescherming van het geheugen van de verschillende jobs tegen elkaar.

De exchange jump lijkt enigszins op wat bij andere computers bestaat onder de naam 'interrupt'. "Echte" interrupts kent de centrale processor niet, ze komen trouwens in deze computer helemaal niet voor.

2.2 De computer - perifere verwerkingseenheden

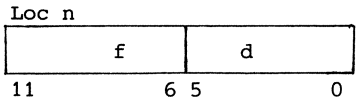
In tegenstelling tot de centrale processors, die samen het centrale geheugen gebruiken, hebben de perifere verwerkingseenheden (PPU's) elk een eigen geheugen van 4k woorden van 12 bits. Hoewel de PPU's zich op het eerste gezicht gedragen als zelfstandige , onafhankelijke computer(tje)s, is dit in feite niet het geval. De PPU's vormen per groep van 7 of 10 stuks een bij elkaar behorend geheel. Vandaar tussen twee haakjes de aantallen PPU's die een computer kan hebben: 10, 14, 17 of 20. De registers en de 4k geheugen-modules zijn er per PPU; per groep is er één processor. De PPU's denke men zich hierbij in een ring geschakeld, de zg. barrel; alle PPU's passeren achtereenvolgens de processor (slot), waarbij een actie uitgevoerd wordt. Op deze manier verwerkt de processor iedere 100 nsec een PPU; de effectieve cyclustijd van een PPU wordt hierdoor 1 µsec.



2.2.1 Adresseringswijze

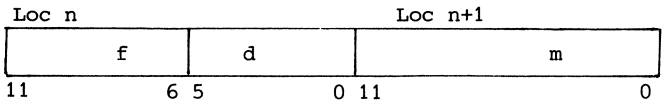
In verband met hetgeen verderop in dit verhaal volgt over het besturings-systeem, waarin de PPU's een belangrijke rol zullen spelen, zal hier wat dieper dan het geval was bij de CPU worden ingegaan op de eigenschappen van de PPU vanuit programmeringsoogpunt.
De PPU heeft één rekenregister, het A register, groot 18 bits. Er zijn twee instructieformaten: één- en tweewoords. De éénwoordsinstructies zijn aldus

opgebouwd:



Hierbij is f de instructiecode. De betekenis van d is afhankelijk van de soort instructie, dus van de waarde van f. Zo kan d de operand zelf zijn, dat wil zeggen één van de getallen 0 - 63, of het adres van een geheugenplaats, dat wil zeggen locatie 0 - 63. Alleen deze geheugenplaatsen zijn direct adresseerbaar met een enkellengte instructie, en heten daarom 'direct cells'.
Ook zijn er nog instructies waarin d een 'indirect address operand' is. Hierbij is de inhoud van de geheugenplaats met adres d het adres van de operand.
In sommige instructies kan d nog een andere betekenis hebben, bijvoorbeeld een kanaalnummer.

De dubbellengte-instructie ziet er als volgt uit:



Wederom is f de instructiecode.
Als f een instructie is met 'no address operand', dan is de operand het 18 bits getal, bestaande uit de concatenatie van d en m. 18 Bits operanden heeft men bijvoorbeeld nodig bij instructies die in het centrale geheugen lezen en schrijven.
Als f een instructie is met 'direct address operand', dan is de operand de inhoud van de geheugenplaats (d)+m, dat wil zeggen: het adres m, vermeerderd met de inhoud van de direct cell met adres d. Weer zien we dus dat de geheugenplaatsen 0 - 63 een bijzondere eigenschap hebben: ze kunnen voor indexering worden gebruikt.

Een voorbeeld. Veronderstel alle getallen in dit voorbeeld octaal, en

d = 43
m = 1011

terwijl voorts de inhoud van geheugenplaats 0043 = 0123

de inhoud van geheugenplaats 0123 = 7654
de inhoud van geheugenplaats 1134 = 3521.

Er bestaan vijf instructies met de betekenis "laad het A register", voor elk van de bovenomschreven adresseringswijzen één. Bij de bovenveronderstelde waarden van d en m, en inhoud van enkele geheugenplaatsen geeft de volgende tabel de inhoud van register A na voltooiing van de instructie:

TYPE OPERAND	INSTRUCTIE LENGTE	INSTRUCTIE	OPERAND	A-REGISTER
'no address'	enkel	14dd	d	000043
	dubbel	20dd mmmm	dm	431011
'direct address'	enkel	30dd	(d)	000123
	dubbel	50dd mmmm	(m+(d))	003521
'indirect address'	enkel	40dd	((d))	007654

2.2.2 Instructieverzameling

De instructieset van de PPU's is eenvoudiger dan die van de CPU's. Wat betreft de arithmetische bewerkingen zijn alleen optellen, aftrekken, schuifopdrachten en logische opdrachten beschikbaar. Daartegenover staat dat de PPU's in staat zijn daadwerkelijk met hun buitenwereld te communiceren. Ze hebben toegang tot het centrale geheugen en de kanalen. Voorts kunnen ze de handelingen van de CPU's sturen door exchange jumps te doen uitvoeren. Zoals reeds eerder gezegd heeft dit onderscheid met de faciliteiten van de CPU's te maken met het vooropgezette gebruiksdoel van de PPU's: het verzorgen van de in- en uitvoer en het verrichten van de besturingstaken.

2.2.3 Kanalen

De PPU's hebben toegang tot de kanalen, waarvan er 12, 18, 21 of 24 aanwezig zijn (in geval van resp. 10, 14, 17 of 20 PPU's), en wel met dien verstande, dat iedere PPU kan communiceren via alle kanalen. Met de naam "kanaal" wordt in het algemeen bij computers dikwijls een soort I/O processor aangeduid; hier is het in feite niets anders dan een 15-bit-register, dat gelezen en geschreven kan worden door de PPU's, en door de besturingseenheden (controllers) van randapparaten die aan dat kanaal zijn gekoppeld. Van deze 15 bits zijn er 12 databits, een 'active/inactive flag', een 'full/

empty flag' en een functie/data bit. Via de databits zenden PPU en randapparaat data naar elkaar, en verstuurt de PPU functiecodes naar het randapparaat. Het randapparaat kan deze twee gevallen onderscheiden aan de hand van de waarde die het functie/data bit heeft.
Door het zetten van de 'active/inactive flag' activeert de PPU de partner aan de andere kant van het kanaal. Bovendien kan een PPU aan dit bit zien of het kanaal door een andere PPU wordt gebruikt. Door het aan- of afzetten van de 'full/empty flag' geeft het randapparaat aan dat de door de PPU gestarte lees- of schrijfactie door hem voltooid is.

Behalve voor de communicatie PPU - randapparaat kunnen de kanalen ook gebruikt worden voor het verzenden van data naar elkaar. Dit gebeurt in het Scope besturingssysteem bijvoorbeeld als een PPU data van een schijfgeheugen nodig heeft. Hij leest die dan niet zelf, maar laat het een andere PPU doen, die de data via een kanaal aan hem doorgeeft.

De PPU's hebben het volgende instructierepertoire voor de communicatie via kanalen. De letters f, d en m hebben dezelfde betekenis als in 2.2.1.

(f)
(64) Spring naar m als van kanaal d de 'active/inactive flag' op staat
(65) Spring naar m als van kanaal d de 'active/inactive flag' af staat
(66) Spring naar m als van kanaal d de 'full/empty flag' op staat
(67) Spring naar m als van kanaal d de 'full/empty flag' af staat
(70) Lees register A vanuit kanaal d
(71) Lees buffer met lengte A vanuit kanaal d naar adres m en verder
(72) Schrijf de inhoud van register A naar kanaal d
(73) Schrijf buffer met lengte A vanuit adres m en verder naar kanaal d
(74) Zet de 'active/inactive flag' op
(75) Zet de 'active/inactive flag' af
(76) Zet functie/data bit en schrijf register A naar kanaal d
(77) Zet functie/data bit en schrijf m naar kanaal d

Hoe precies in- en uitvoer van en naar een randapparaat zal dienen te verlopen wordt natuurlijk bepaald door de eigenschappen van de besturingseenheid van dat randapparaat. Het lezen van een buffer zou dan in een theoretisch geval bijvoorbeeld als volgt kunenn verlopen:
- Test of het kanaal niet actief is. Dit kan gedaan worden met de (64) of (65) instructie.
- Bepaal of het randapparaat in een zodanige toestand is dat het een buffer data kan overzenden. Dat gaat met een "schrijf functie" instructie,(76) of

of (77), gevolgd door een "activate" instructie (74).
- Wacht op de reactie van het randapparaat door in een lus te testen op het opkomen van de 'full/empty flag' met instructie (66) of (67).
- Lees de reactie van het randapparaat met een (70) instructie en "disconnect" het kanaal (75).
- Als het randapparaat gemeld heeft dat het gereed is voor het verzenden van de data buffer, geef dan opdracht daartoe door de daarvoor bestemde functie over te sturen, (76) of (77), het kanaal te activeren (74), en het data-transport te starten (71).
- Tenslotte wordt na beëindiging van het transport het kanaal weer vrijgegeven, instructie (75).

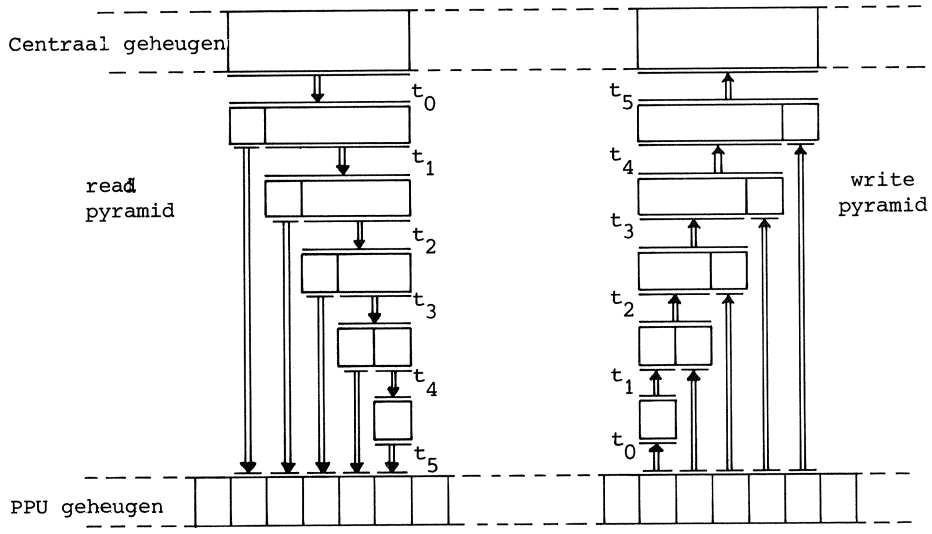
In het algemeen is in het Scope besturingssysteem de gang van zaken anders dan in dit voorbeeld, omdat deze procedure in bepaalde omstandigheden problemen veroorzaakt.
Stel bijvoorbeeld het geval dat PPU_n op de boven omschreven wijze test of een kanaal vrij is, een functie schrijft en het kanaal activeert, terwijl gedurende deze stappen PPU_m ook test of het kanaal vrij is. Ook hij zal dan denken dat hij zijn gang kan gaan, een functiecode schrijven en het kanaal activeren.
Oplossing wordt gegeven door in het besturingssysteem een andere manier van kanaalreservering te gebruiken. In de beschrijving van het systeem wordt hierop teruggekomen.

De peripheral processors kennen geen interrupts. Voor de (71) en (73) instructie wordt de volgende instructie pas uitgevoerd als het gehele transport voltooid is. Voor de andere gevallen dat een instructie een randapparaat start, moet programmatisch op de 'full/empty flag' worden getest om te zien of de actie voltooid is.

2.2.4 Toegang tot het centrale geheugen

De PPU's hebben toegang tot het gehele geheugen van de centrale verwerkingseenheden. Lezen en schrijven gebeurt met eenheden van één CM woord en vijf opvolgende PPU geheugenwoorden. De PPU's hebben instructies voor het transporteren van een CM woord of een buffer van CM woorden. In beide gevallen wordt pas verder gegaan met de volgende instructies als het gehele geheugentransport voltooid is.

Het samenvoegen van een CM woord uit 5 PPU woorden en het opsplitsen van een CM woord in 5 PPU woorden gebeurt via de zg. 'read-' en 'write pyramid', waarvan er voor elke barrel van 7 of 10 PPU's één aanwezig is. Hierbij wordt voor de 'read pyramid' op iedere "klokpuls" van de PPU, dat is dus iedere microseconde, een woord uit het PPU geheugen gelezen, en samengevoegd met wat al eerder gelezen is. Bij het schrijven in CM vindt iets dergelijks plaats. Transport van een CM woord duurt aldus 5 µsec. De pyramides kunnen echter de geheugentransporten van vier PPU's tegelijk verwerken, zoals ook uit de beschrijving en onderstaande tekening duidelijk zal zijn.



2.3 De computer- slotopmerkingen

De hierboven gegeven beschrijving van de 'lower Cyber series' computers is noodzakelijkerwijze beperkt gebleven. Niet besproken is een aantal, op zichzelf best interessante onderwerpen als gelijktijdige toegang tot verschillende centraal-geheugen-banken, het 'interlock register', het 'distributive data path' via welk de PPU's direct toegang tot ECS hebben, gelijktijdige verwerking van een aantal instructies door de verschillende functionele eenheden in de Cyber 74, enzovoort. Ook is terwille van de eenvoud de waarheid op enkele detailpunten geweld aangedaan. Wat resteert is bedoeld een basis te zijn, waarmee de hierna volgende beschrijving van het besturingssysteem begrepen kan worden.

3. HET BESTURINGSSYSTEEM

Het Scope besturingssysteem voor de Cyber computer is een multiprogrammeringssysteem; een aantal jobs wordt dus gelijktijdig verwerkt. Het besturingssysteem bestaat uit twee gedeelten: programma's die uitgevoerd worden door een centrale processor en daarom een gedeelte van het centrale geheugen bezetten, samen CPMTR geheten, en een gedeelte dat bestaat uit PPU programma's. De kern is het PPP programma MTR, waarvoor één van de PPU's, PP0, vast gereserveerd is.
Ook PP1 is vast toegewezen, namelijk aan het programma DSD (dynamic system display), dat de operateursconsole bestuurt. De overige PPU's, wel aangeduid met de naam 'pool-PPU's', zijn in principe vrij. Aan hen wordt door MTR als deze er behoefte aan heeft, opdracht gegeven een PPU programma uit te voeren. Alle pool-PPU's zijn gelijkwaardig; een programma zal nu eens in de ene, dan weer in een andere PPU verwerkt worden. Ook zal vaak een PPU programma door een aantal PPU's tegelijk verwerkt worden.

We hebben dus de volgende situatie:

- In PP0 de monitor MTR;
- In de andere PPU's een steeds wisselend aantal, steeds weer andere programma's, uitgevoerd in opdracht van MTR;
- In CM een gedeelte van de monitor dat door de CPU's wordt uitgevoerd, genaamd CPMTR;
- In CM een aantal gebruikersprogramma's.

Op welke wijze bestuurt MTR nu dit geheel, en hoe vindt de communicatie tussen deze processen onderling plaats? Met deze vraag zullen wij ons in de rest van dit verhaal bezighouden.

3.1 Tabellen in het centrale geheugen

Voordat de wijze van communicatie voor de verschillende gevallen wordt uiteengezet dient herinnerd te worden aan een bijzondere eigenschap van het centrale geheugen. CM is namelijk de enige plaats in de computer waar data kan worden opgeslagen die voor alle processoren, CPU's en PPU's, toegankelijk is. Een stuk van het centrale geheugen is daarom gereserveerd voor tabellen. Voorbeelden zijn de 'File Name Table', waarin de gegevens van alle files in het systeem worden opgeslagen; de 'Record Block Reservation Table', waarin geadministreerd wordt welke stukken van het schijfgeheugen door files bezet worden; de 'Peripheral Job Table', waarin MTR bijhoudt met welke programma's

de PPU's bezig zijn; de 'Control Point Areas', waarin de status wordt bijgehouden van de jobs die in behandeling zijn; de 'PP Communication Areas', via welke MTR en de pool-PPU's met elkaar praten.
Het laatste voorbeeld zal in het verdere verhaal nog ter sprake komen.

3.2 Communicatie gebruikersprogramma - besturingssysteem

Zoals in 2.1 uiteengezet kan de CPU tijdens de verwerking van een gebruikersprogramma slechts geheugenplaatsen binnen het voor dit programma beschikbare stuk geheugen bereiken. De centrale processor heeft ook geen andere mogelijkheden om het systeem duidelijk te maken dat hij iets gedaan wil hebben. (Weliswaar bestaat sinds enkele jaren de 'central exchange jump, waarmee een CPU de controle kan overdragen aan CPMTR, maar aan de manier van werken, die gebaseerd is op het bovenstaande, is in principe niets veranderd.) Als nu een job een actie door het systeem verricht wil hebben, bijvoorbeeld omdat in- of uitvoer noodzakelijk is, of omdat de job een deeltaak heeft voltooid, dan wordt dit kenbaar gemaakt door een verzoek te schrijven op geheugenplaats 1, uiteraard van het aan de job toegewezen stuk geheugen (een zg. 'RA+1 request'). Deze geheugenplaats is voor dit doel gereserveerd. MTR leest regelmatig van alle jobs geheugenplaats 1 uit. Het fysieke adres kan hij vinden met behulp van de Control Point area van de job. Als regel zal de inhoud van deze geheugenplaatsen nul zijn, behalve als de job een RA+1 request gedaan heeft. De meest significante 18 bits van het woord bevatten dan de drieletterige naam van het programma, een PPU programma of een programma in CPMTR, waarvan de job wil dat het uitgevoerd wordt. De rest van het woord bevat aanvullende informatie. Als MTR een RA+1 request vindt, dan zal hij een CPU een exchange jump laten uitvoeren naar CPMTR. Deze bekijkt het verzoek nauwkeuriger, voert het indien mogelijk uit, of geeft de nodige verzoeken aan MTR. Als met de uitvoering van het RA+1 request wordt begonnen wordt dit aan de job medegedeeld door de inhoud van zijn geheugenplaats 1 weer nul te maken.
Een gebruikersprogramma kan bijvoorbeeld verzoeken de volgende programma's te doen uitvoeren:

- TIM. Dit programma geeft tijd of datum. Het verzoek wordt geheel door CPMTR afgehandeld.
- CIO. Dit is het algemene in- en uitvoer-commando. In het algemeen zal zo'n commando acties nodig maken van een randapparaat, en dus de nodige aanroepen door CPMTR van MTR veroorzaken.

3.3 Communicatie MTR - CPMTR

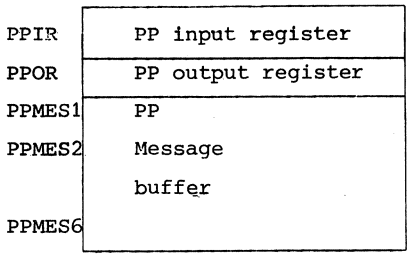
Voor het aanroepen van MTR door CPMTR is er een 4-woords commandobuffer in CM aanwezig, T.TRRS. Als er een plaats in deze circulaire buffer vrij is, dan wordt het verzoek erin geplaatst, anders moet CPMTR wachten tot er een plaats vrij komt. Regelmatig wordt de buffer door MTR geraadpleegd om te zien of er opdrachten in staan.

Het omgekeerde, aanroepen van CPMTR door MTR, vindt op dezelfde manier plaats als waarop pool-PPU's een CPMTR aanroep doen (zie 3.5).

3.4 Opdrachten van MTR aan een pool-PPU

PPU's krijgen hun opdrachten slechts van MTR. Hiertoe is er voor iedere PPU een stuk geheugen in CM gereserveerd, de zg. PP Communication Area, waarvan de opbouw hieronder is gegeven. Als MTR een bepaald PPU programma

PP Communication Area

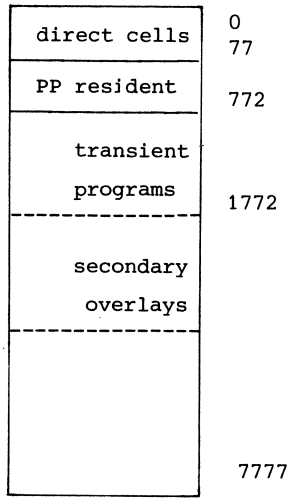


uitgevoerd wil hebben, dan plaatst hij de naam van het programma in het input register, PPIR, van een vrije PPU. Van een vrije PPU is het PPIR nul.

Let wel: er is geen hardware-mogelijkheid voor MTR om een PPU te dwingen tot een bepaalde actie! Dat het bovenstaande functioneert komt doordat dat er stringente afspraken bestaan over wat een PPU wel en niet mag doen. Eén van die afspraken is, dat als een PPU vrij is, hij een programma dient uit te voeren dat regelmatig zijn PPIR bekijkt om te zien of er een actie van hem verwacht wordt.
Als voor een PPU programma iets gedaan moet worden door MTR of CPMTR dan geeft hij dat te kennen door een verzoek in zijn PPOR te plaatsen.
Daarna moet hij wachten tot het verzoek is uitgevoerd, hetgeen kenbaar ge-

maakt wordt door de meest significante 12 bits van het PPOR nul te maken.

Voor al deze voorgeschreven acties van de PPU's zijn er subroutines in de PPU's die, ongeacht het uitgevoerde programma, altijd aanwezig zijn, en door deze programma's gebruikt (moeten) worden. Het 4k geheugen van de PPU is daarom als volgt ingedeeld:



Van de direct cells is een aantal voor vaste doeleinden gereserveerd.
Zo bevat locatie 74 het CM adres van PPIR van de betreffende PPU en locatie 75 het adres van zijn PPMES.
PP Resident bevat een aantal subroutines, waarvan er enkele hierboven reeds zijn aangeduid. De volgende opsomming is meer, zij het niet geheel, volledig. Gebruikt zijn de namen die ook gebruikt worden bij het schrijven van PPU programma's.

R.IDLE Dit is het programma dat wordt uitgevoerd als de PPU vrij is. Regelmatig wordt gekeken naar de waarde van PPIR; echter met intervallen om te voorkomen dat het CM-leesmechanisme via de 'read pyramid' verstopt raakt.

R.OVLJ Laadt een PPU programma en start dit.

R.OVL Laadt een overlay van een PPU programma.

R.READP Leest via een kanaal een buffer data van een andere PPU, die toegang heeft tot het schijfgeheugen (de 'stack processor').

R.WRITEP Schrijft data via een kanaal naar de stack processor, die het naar het schijfgeheugen brengt.

R.RAFL 'request access to control point field length'. Nodig in verband

met mogelijke conflicten tussen enerzijds de 'storage move' routine in CPMTR, die het stuk geheugen van een job wil verschuiven, en anderzijds een PPU die in dit stuk geheugen wil lezen of schrijven. Zie verder 3.4.1 .

R.TAFL 'terminate access to control point field length'. Zie 3.4.1.

R.MTR Doet een verzoek aan MTR of PCMTR, door dit in PPOR te plaatsen. Wordt gevolgd door een R.WAIT .

R.WAIT Wordt uitgevoerd na een R.MTR . Bepaalt of het verzoek bestemd was voor MTR danwel CPMTR. Wacht voorts tot de meest linkse 12 bits van PPOR nul zijn gemaakt, ten teken dat het verzoek is uitgevoerd.

R.RCH Reserveert een kanaal. Deze routine doet dat door de MTR-functie M.RCH aan te roepen.

R.DCH Geeft gereserveerd kanaal vrij.

In bovenstaand overzicht komen twee interessante gevallen voor, namelijk de manier waarop PPU's toegang krijgen tot het stuk geheugen dat aan een job is toegewezen, en de manier waarop een kanaalreservering tot stand komt. Het laatste gebeurt niet rechtstreeks met de daarvoor aanwezig instructies, zoals ook reeds in 2.2.3 werd aangestipt.

3.4.1 Toegang tot een control point

Het lezen of schrijven door een PPU in het stuk geheugen dat aan een job is toegewezen dient met enige omzichtigheid te geschieden.
Het is immers mogelijk dat dit stuk geheugen juist op hetzelfde moment verplaatst wordt (door CPMTR). Dit conflict tussen het PPU programma en de 'storage move' routine wordt als volgt opgelost. Er is in de 'control point area' van de job een 'storage move flag', die door de 'storage move' routine voor de duur van de geheugenverplaatsing wordt opgezet. Voorts heeft de PPU een zg. 'field access flag'. De routine R.RAFL gaat nu als volgt te werk:

- De 'field access flag' wordt gezet. Daardoor zal vanaf dit moment geen 'storage move' voor de betreffende gebruikersjob meer gestart worden.
- Getest wordt of de 'storage move flag' voor de betreffende job op staat. Zo ja, dan wordt de 'field access flag' afgezet, waarna het geheel herhaald wordt.
- In het andere geval was er geen 'storage move' aan de gang. Omdat de 'field access flag' inmiddels op staat zal er ook geen 'storage move' gestart worden.De PPU kan dus rustig in het stuk geheugen lezen en schrijven.

Als hij klaar is dient hij met een R.TAFL aanroep de 'field access flag' weer af te zetten.

3.4.2 Het reserveren van kanalen

In 2.2.3 is er op gewezen dat de hardware instructies die voor het reserveren van kanalen beschikbaar zijn, geen afdoende beveiliging bieden tegen het reserveren van een kanaal door meerdere PPU's tegelijk. Daarom wordt in Scope de reservering van een kanaal niet door de PPU routine zelf gedaan, maar, via een daartoe strekkend verzoek, door MTR. Deze houdt in CM een reserveringstabel van de kanalen bij, en reserveert het kanaal. Het vrijgeven van een kanaal gebeurt wèl door de PPU zelf. Daartoe kijkt hij eerst in de tabel of hij dit kanaal heeft toegewezen gekregen, als dit zo is dan 'disconnect' hij het kanaal (instructie (75)), en werkt de reserveringstabel bij.

3.5 Communicatie van een PPU programma met MTR en CPMTR

Als een PPU programma een verzoek wil richten tot MTR of CPMTR dan plaatst hij dit verzoek in zijn output register, PPOR. Dit gebeurt met de 'resident' routine R.MTR . De 'resident' routine die daarna uitgevoerd wordt, R.WAIT, plaatst bovendien het beginadres van de betreffende PP communication area (oftewel het adres van PPIR) in één van de twee geheugenplaatsen PPID en PPIP, welke zich ook bevinden in het voor tabellen gereserveerde gedeelte van het centrale geheugen. De keuze PPID of PPIP is afhankelijk van de bestemming van het verzoek voor respectievelijk CPMTR of MTR. Bovendien wordt, als het verzoek voor CPMTR bestemd was, een exchange jump opgewekt.
Hiermee wordt bereikt dat MTR en CPMTR niet regelmatig alle PPOR's behoeven na te lopen om te zien of er een verzoek staat te wachten, maar kunnen volstaan met het bekijken van de inhoud van resp. PPIP en PPID. Als er echter een verzoek is, dan dienen na afhandeling ervan wél alle PPOR's nagegaan te worden, omdat ieder volgend verzoek de oude waarde van PPID resp. PPIP overschrijft. Voltooiing van de afhandeling van het verzoek wordt de PPU bekend gemaakt door de meest linkse 12 bits van PPOR nul te maken. R.WAIT wacht hierop. De overige bits van PPOR kunnen na afloop informatie over de afhandeling bevatten.

Bespreken van alle opdrachten die aan MTR en CPMTR gegeven kunnen worden is in dit bestek niet mogelijk en niet zinvol. Het volgende moge een

idee geven.

3.5.1 Enkele opdrachten voor CPMTR

M.ICE (Initiate Central Executive). Hiermee kan verzocht worden diverse vaste CPU programma's uit te voeren, bijvoorbeeld de hiervóór enkele malen genoemde 'CM storage move' routine, of de "Integrated Scheduler', die de volgorde van bewerking van de in afhandeling zijnde jobs bepaalt.

M.RCH Reserveer een kanaal.

M.SLICE Beeindig de 'time-slice' van een CPU programma.

3.5.2 Enkele opdrachetn voor MTR

M.DFM Genereer een dayfile boodschap. De dayfile bevat een overzicht van de handelingen op stuurkaartniveau die ten behoeve van de uitvoering van een job zijn verricht.

M.DPP 'drop PPU'. Hiermee meldt de PPU zich klaar.

M.RPJ Verzoek om een PPU programma te doen uitvoeren.

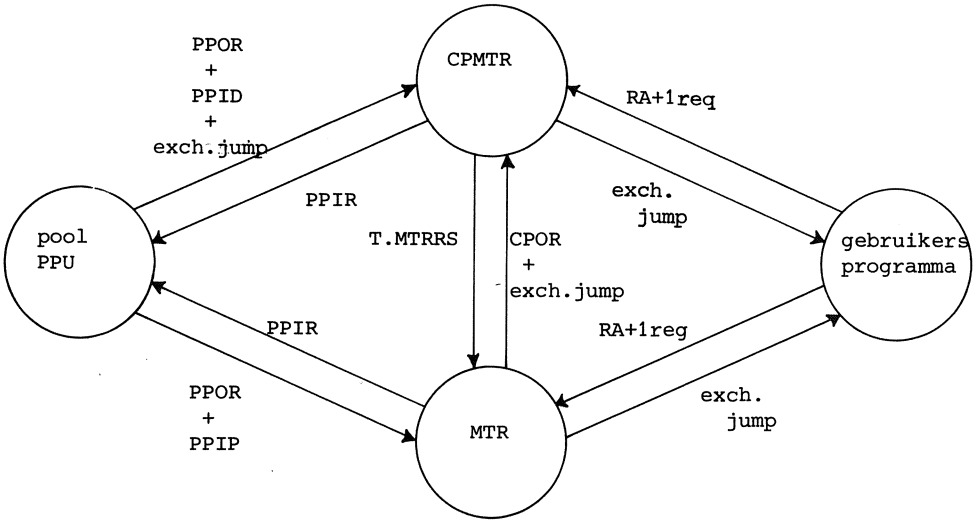
M.PASS Synchronisatie functie. Het geven van deze opdracht, gevolgd door een R.WAIT laat een PPU programma pas verder gaan nadat een ander PPU programma hem heeft doorgestart.

M.PATCH Maak een verandering ('patch') in MTR.

M.KILL Veroorzaakt een 'bad monitor request'. De machine gaat down. Deze functie wordt door een PPU aangeroepen als hij bijvoorbeeld iets vindt in een commiunicatietabel in CM wat geen betekenis heeft.

3.6 Communicatie overzicht

Het volgende diagram geeft de verschillende besproken wijzen van communicatie nog eens schematisch weer.


CPMTR
PPOR
+
PPID
+
exch.jump
RA+1req
PPIR
exch.
jump
pool
PPU
T.MTRRS
CPOR
+
exch.jump
gebruikers
programma
PPIR
RA+1reg
PPOR
+
PPIP
MTR
exch.
jump

MATHEMATICAL CENTRE SYLLABUS 32, 1976, VI, 101-112

SOME ASPECTS OF THE EFFICIENCY OF SYSTEM IMPLEMENTATION LANGUAGES

B.A. WICHMANN
National Physical Laboratory, UK

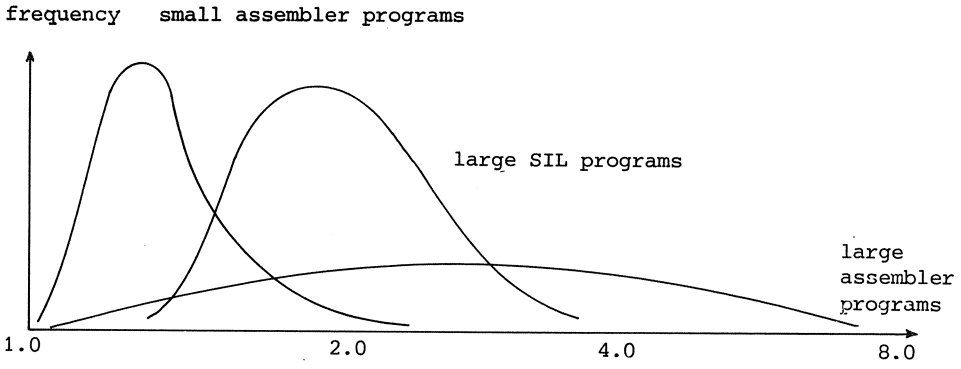
0. INTRODUCTION

What is a System Implementation Language or SIL? The simplest definition is that it is a high level language designed as a replacement for an assembler. All software, particularly system software, which has been traditionally written in an assembler should be capable of being written in the SIL. Many systems programmers will be horrified that an operating system or compiler is written in a high level language because they suspect it will not be "efficient" enough. This is to misunderstand the nature of SILs and indeed the typical efficiency of many assembler programs. In a SIL one attempts to combine the obvious advantages of a high level language with the degree of control over the machine that an assembler gives. Many such languages are highly successful in this aim, and I shall try to explain why.

1. EFFICIENCY

Few people question the "efficiency" of assembler-written programs and yet no non-trivial program can be expected to be perfect. Highly tuned programs can be very awkward owing to inflexibilities in their structure, which can make maintenance almost impossible. I would recommend as an exercise that the readers of this note check the efficiency of one assembler program. On KDF9, after nearly 10 years of use, our assembly code translator was found to be half the speed that it should be owing to unnecessary conversion of characters on input. Assembler programs of any size, especially if they have been heavily modified, cam become very poorly structured. In consequence, further modifications are usually patched in so that any advantages of a new algorithm or data structure cannot be exploited. Hence the programs become needlessly inefficient.

If one could measure the efficiency of a large number of programs written in conventional assembly languages and SILs, I think one would get the following histogram:



Efficiency compared with optimal hand coding (conjecture).

2. HOW HIGH LEVEL?

How is it that programs written in a SIL can be relatively efficient whereas traditional high level languages are rarely very efficient? In designing a SIL, based upon ALGOL 60, say, one looks critically at each feature of the language and decides whether it is worth including and if not, what substitute would be acceptable. Unfortunately the cost of implementing a langauge feature can depend upon the underlying hardware - resulting in a machine-dependent SIL.

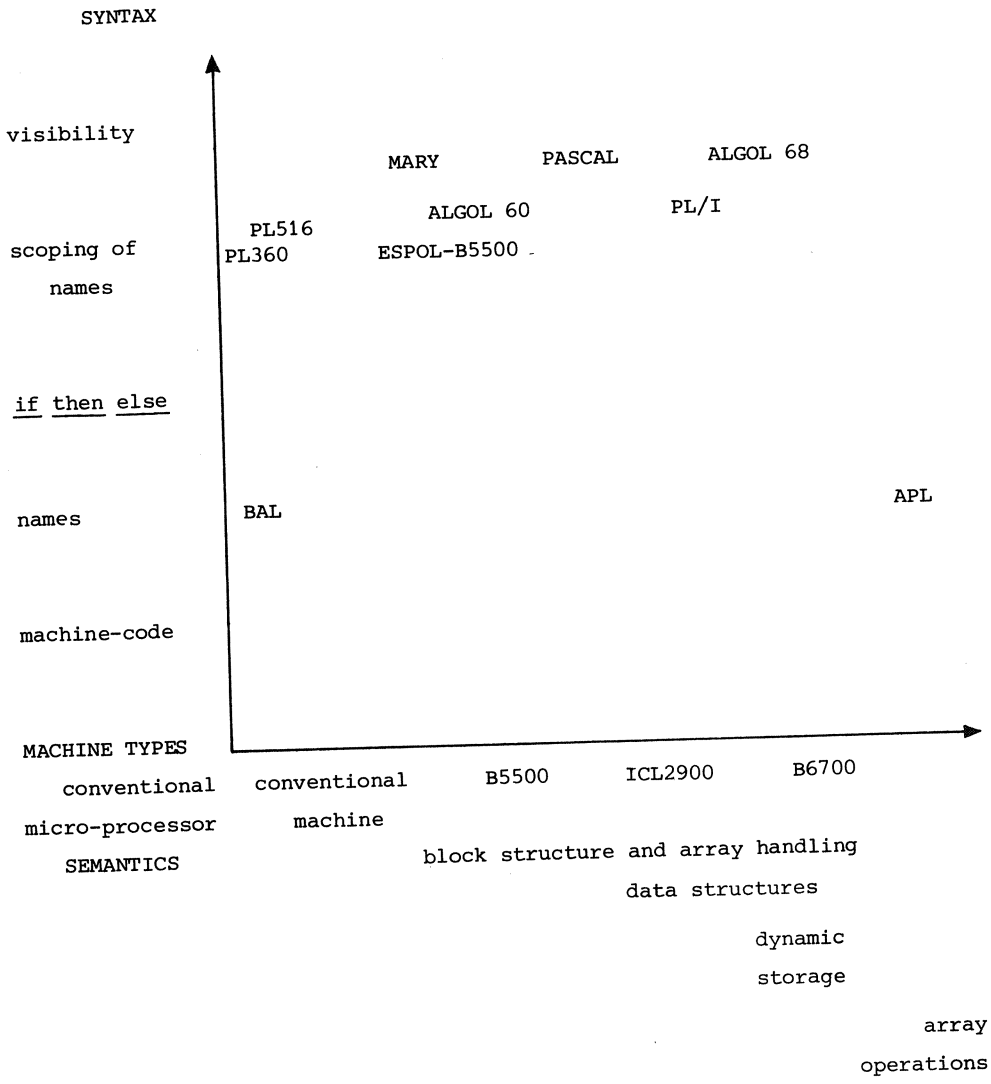
Some high level language features are very easy to implement and are also efficient. The most obvious example is good control structures such as if then else, case statement, iterative loops, while etc. In many cases it is difficult to see why they have not been added to conventional assemblers since the advantages in program clarity is so great. The "overhead" introduced by this facility amounts to no more than 5% and results from a compiler generating extra unconditional jumps (jump over else). This could be avoided by reordering the code (by optimization) as could be done by the assembler programmer.

The manner in which additional features are provided apart from control structures depends upon a vital decision. Does one provide a high-level assembler or a true System Implementation Language? A high-level assembler is overtly machine dependent and is really an assembler which supports structured programming. Examples of this school are PL/360 (360/370 machines [1]), PL516 (Honeywell Series 16 [2]) and Babbage (GEC 4080 [3]). The true SIL hides some machine features to provide a more elegant language at some cost in efficiency. Examples of this approach are BCPL [4], BLISS [5], PASCAL [6] and MARY [7].

Another facility which can be added at no cost at all is to allow scoping of names by means of a block structure. The block structure need not incur any storage allocation at all since the storage can be allocated at compile-time as occurs in CORAL 66 (a real time language which is a dialect of ALGOL 60 [8]). The scoping allows the programmer to protect temporary variables from being overwritten by misuse at an outer level - the code can be structured better. In large software systems it is necessary to have better control over the access to names than the ALGOL-like block structure can provide. As an example, two disc utility procedures, P and Q may be of general use and hence are declared at a global level, but access to working buffers and pointers that they have in common may be disastrous. A technique used in LIS [9], a SIL designed at CII, is to have units of visibility to control the access to the correct extent.

Another way of expressing the same point is to classify languages by means of their syntax and semantics. Control structures increase the syntactic structure of the language but do not increase the semantics. The semantics provided in the basic hardware varies from nearly full ALGOL 60 (B6700) to an extremely primitive order code of some microprocessors. This is illustrated by the syntax-semantics diagram below. If the language is further to the right (i.e. has more semantics) than the machine, then some loss in efficiency is likely.

Diagram of the Syntax and Semantics of SILs



Dynamic allocation of storage and the generalized handling of arrays does incur a penalty on conventional machines. Hence the high-level assemblers on conventional machines provide arrays only by means of explicit use of index registers (without bound checking). If dynamic storage allocation is adopted, then various compromise solutions are possible. For instance, in the B5500 hardware and the design of BCPL one is restricted to accessing local or global variables - any intermediate levels cannot be accessed. This simplifies the organisation of stack storage. The majority of SILs (as opposed to high level assemblers) have a stack and hence permit recursion.

The advantage of a stack is that code is naturally reentrant and that the total data storage required is less than with static storage. For instance, in the EMAS operating system written in IMP, 17848 bytes of declared variables require only 3K in stack space [10].

The disadvantage of stack storage is the overhead on conventional machines in updating the stack pointers on procedure entry and exit. Several studies have shown that the overhead on procedure call in compilers written in SILs can be 25%. A test has been designed to illustrate this by timing a highly recursive procedure called Ackermann's Function. A summary of some of the results of this test is as follows:

Recursive subroutine linkage overhead

Machine range	Language	Instructions per call
IBM 360/370		
	Assembler	6
	IMP (Edinburgh)	19
	ALGOL 60 (Edinburgh)	21
	PL/I (OPT)	≃ 61
	ALGOL W	≃ 74
	ALGOL 60 (Delft)	≃ 142
	PL/I (F)	≃ 212
	ALGOL 60 (F)	≃ 820
CDC 6000		
	PASCAL	38.5
	Mini-ALGOL 68 (Amsterdam)	51
	ALGOL 68	≃ 122
	SIMULA	≃ 770
	ALGOL 60	≃ 890
ICL 1900		
	Assembler	7.5
	ALGOL 68-R	28
	PASCAL	32.5
	ALGOL 60 (Manchester)	33.5
	ALGOL 60 (ICL)	≃ 120

In some cases, the number of instructions per call has been estimated from the time taken (and should be within 20% of the correct value). For further details, see WICHMANN [13].

3. DATA STRUCTURES

Languages of greater semantic power than ALGOL 60 provide the ability to declare and manipulate records or data structures. This is a substantial advance for writing system software since it allows data to be structured in the same way that control structure aids the intelligibility of code. Unfortunately great care must be taken in putting such features into a SIL. In ALGOL W and SIMULA, data structures imply the use of a general storage management scheme which could make it virtually impossible to write operating

systems in those languages. On the other hand, a very rich data structuring capability is available in PASCAL without complex store management. In LIS one has the option of declaring data structures within an area in which one can provide ones own storage management package.

Apart from the storage management for data structures, the actual form and access to the structures is critical. Most high-level assemblers, if they have extensive structuring at all, will be confined to those units of storage which are directly addressible in an obvious manner. PL360 and PL516 only have arrays of the basic hardware formats whereas Babbage has some structures corresponding to the more elaborate addressing facilities of the GEC 4080.

The machine independent languages pose a difficult design problem since nothing can be assumed about the underlying machine. In BCPL, the only element of storage is the word which must be explicitly packed and unpacked by the programmer if required. In CORAL 66, one can specify part-word items at the bit level although this may be very expensive in accessing time and space. The more recent languages are based upon PASCAL and ALGOL 68. Good declarative information, which in PASCAL includes ranges of values for enumerated types, allows an optimizing compiler to generate high quality code. In LIS, the syntactic form of access to a data structure is independent of its physical layout. This allows a good compiler to choose an optimal layout or the user to specify a layout to conform to some hardware requirement.

4. CONCLUSIONS

By good language design, it is possible to produce a System Implementation Language which is very efficient. The efficiency cannot match a hand-coded assembler program, but when one takes into account programming and maintenance costs of software, the cost advantages of SILs are considerable. Machine dependent SILs and particularly the high-level assemblers, have the edge in efficiency because of the ability to exploit every machine feature - often in a "well structured" fashion. A high degree of machine independence is possible by isolating the machine dependent parts (which are inevitable in system programs). Optimizing compilers can compensate for a small degree of mismatch between the language and the machine. For an exposition of optimizing techniques applied to a SIL (the PDP11 version of BLISS), see

WULF e.a. [12].

Almost all manufacturers and many software houses are using SILs for major software systems. To understand the SIL approach, one should use such a system experimentally. Most computers have such a language or one can even implement BCPL in a few man months. For those in the UK, the British Computer Society holds regular meetings on SILs at one of its Specialist Groups. At the international level, the International Federation for information Processing has a Working Group devoted to the study of these languages (WG2.4). Bodies such as CCITT, the EEC and the US Department of Defense are actively investigating the design and use of SILs. The technical journal which covers this area best is probably Software - Practice and Experience.

5. SOME EXAMPLES

To give some idea of the power of expression of some SILs a few examples are given:-

PL360 - The first high-level assembler.

```
                                                    comment

procedure Magicsquare (R6);                 R6 contains the return address
begin
short integer nsqr;                         half-word integer
integer register n syn RO, ..               n names register RO
:
ij := j shll 2 + x;                         4 * j + x
:
if i > n then i := i-n;
end
```

PL-516 A high level assembler for a mini computer.

```
constant m = -5;
integer i;
procedure  perle;
integer j, j1, j3;
begin
   when i nonzero then
      begin
         for j = m do
            s[j] = neg j;
         i = zero;
         goto rose
      end
end;
```

BCPL

A machine-independent SIL, which is a typeless language. Available on a large number of machines.

```
Transfer (x) be
$(Let a, b = dvece, dvecs
  Let k, n = 0, 0
  Test h1!(h4!x) = s. number
       Then k, n := s. in, h2!(h4!x)
          Or $(k, n := s.lp, ssp
                load (h4!x) $)
  Unless h5!x = 0 Do step := evalconst (h5!x)
 :
 $)
```

BLISS

Another typeless SIL, available on PDP10 and PDP11 computers.
It is an "expression language" and does not contain any goto.

```
function locspacemgr (tog, numb, base) =
     if tog then
            begin
            if (space ptr @. space ptr +. number) > 10000 then
                     return error ();
            space +. space ptr -. numb
            end
else
   ...
```

PASCAL

A high level language, but with some careful restrictions to make it efficient on conventional machines.

```
type days = (mon, tues, wed, thur, fri, sat, sun);
     workd = mon .. fri;
     letter = 'a' .. 'z';

var day : days;
    case day of
       mon, tues: clear in tray;
       wed, thur: do work;
       fri:       tidy in tray;
       sat, sun:  holiday
    end
```

PL/S

IBM's System Language used for writing the virtual memory software.
Not available to customers.

```
DCL ARY(5) FIXED (31) INIT (0,4,8,12,16);
DCL 1 STRUCT
      2 A CHAR (3)
        3 B CHAR (2),
        3 C CHAR (2),
      2 D CHAR (4);
.
.
.
IF CODE = 0 THEN
   DO;
   CALL PRINT (OUT);
   END;
ELSE
   RETURN;
```

REFERENCES

[1] WIRTH, N., *PL360, a programming language for the 360 computer*, JACM 15 (1968) 37-74.

[2] BELL, D.A. & B.A. WICHMANN, *An ALGOL-like assembly language for a small computer*, Software-Practice & Experience 1 (1971) 61-72.

[3] *Babbage Language Manual*, GEC Computers Publication 530/3/TS/2364, 1975.

[4] RICHARDS, M., *The portability of the BCPL compiler*, Software-Practice & Experience 1 (1971) 135-146.

[5] WULF, W.A., D.B. RUSSEL & A.N. HABERMANN, *BLISS: A Language for systems programming*, CACM 14 (1971) 780-790.

[6] WIRTH, N., *The programming Language PASCAL*, Acta Informatica 1 (1971) 35-63.

[7] CONRADI, R. & P. HOLAGER, *MARY textbook*, Trondheim, 1974.

[8] WOODWARD, P.M., P.R. WETHERALL & B. GORMAN, *Official definition of CORAL 66*, HMSO, 1970.

[9] ICHBIAH, J.D., J.P. RISSEN, J.C. HELIARD & P. COUSOT, *The System implementation language LIS*, Technical Report 4549E1/EN, CII.

[10] STEPHENS, P.D., *The IMP language and compiler*, Computer Journal 17 (1974) 216-223.

[11] *Guide to PL/S II*, Publication GC 28-6794-0, IBM, 1974.

[12] WULF, W.A., R.K. JOHNSSON, C.B. WEINSTOCK, S.O. HOBBS & C.M. GESCHKE, *The design of an optimizing compiler*, American Elsevier, 1975.

[13] WICHMANN, B.A., *Ackermann's function: a study in the efficiency of calling procedures*, BIT (to appear), 1976.

MATHEMATICAL CENTRE SYLLABUS 32, 1976, VII, 113-128

JOB CONTROL LANGUAGES

L.G.L.T. MEERTENS

0. INLEIDING

In de tijd toen er al wel computers waren, maar er nog geen sprake was van enigerlei Job Control Langauge (JCL), leverde je gewoon je ponsbandjes of pakje kaarten bij de operateur in, zonodig vergezeld van enkele opmerkingen wat hij eerst moest inlezen of "draaien", of wat gebeuren moest als er iets mis ging. De bedrijfssystemen waren zo eenvoudig, dat je alleen vanuit het heden terugblikkend kunt stellen dat ze er wel degelijk waren.

Al iets ingewikkelder werd het bij de eerste batchverwerkingssystemen. Daar werd het noodzakelijk in de stroom van jobs aan te geven waar de ene ophield en de volgende begon. En voor systemen die een boekhouding bijhielden van verstookte rekentijd, moest de job beginnen met een identificatie van de gebruiker. Voor het overige was de situatie nauwelijks veranderd: het programma had het rijk alleen op de machine, en hoefde niet om de toewijzing van schaarse middelen te vragen - ze waren er of ze waren er niet. In geval van een wat ingewikkelder opgave, zoals: de uitvoer van programma A gebruiken als invoer van programma B, wachtte de gebruiker gewoon tot hij de resultaten van A ontving om een nieuwe job samen te stellen, of, als hij op goede voet met de operateur stond, liet hij dat aan hem over.

Sinds geruime tijd staan operateurs en gebruikers niet meer op goede voet met elkaar; ze verkeren zelfs niet "on speaking terms". Niet uit onmin, maar om de eenvoudige reden dat ze elkaar niet ontmoeten. Met het ingewikkelder worden van de bedrijfssystemen en met name met de invoering van multiverwerkingssystemn, zijn de oorspronkelijke taken van de operateur overgenomen door het bedrijfssysteem. Dat is de nieuwe gesprekspartner van de gebruiker, en aanwijzingen over de verwerking van jobs zijn voor zijn oren bestemd. Die aanwijzingen worden gegeven in een taal: de JCL.

Al is een JCL geen natuurlijke taal, toch zit er een natuurlijk aspect

aan: de faciliteiten die door hedendaagse bedrijfssystemen worden geboden zijn het resultaat van een evolutieproces, het meegroeien met de wensen van een voortdurend groter verscheidenheid aan gebruikers, en met dat proces zijn de JCL's meegeëvolueerd. In het algemeen weerspiegelt een JCL vrij aardig het bedrijfssysteem waarop hij geënt is. Kan dat bedrijfssysteem gekarakteriseerd worden als een orkest van toeters en bellen, dan is de JCL navenant. Deze nauwe correspondentie brengt BARRON & JACKSON [1] ertoe OS/360 JCL te vergelijken met een assembleertaal.

Deze vergelijking roept de vraag op: kunnen de lessen, geleerd bij het ontwikkelen van programmeertalen op een hoger niveau dan dat van assembleertalen, worden toegepast op de ontwikkeling van JCL's?

1. JCL ALS PROGRAMMEERTAAL

Er valt niet alleen een analogie te trekken tussen een JCL en een programmeertaal, maar een JCL is in feite een programmeertaal, zij het dan ook een met doorgaans weinig uitdrukkingsmacht en met een nogal merkwaardige semantiek. Maar bij iedere redelijke definitie van het begrip programmeertaal zal blijken dat een JCL daar ook onder valt. Een programma beschrijft een proces, en in het geval van een JCL-programma wordt als het ware het gedrag beschreven van een operateur-oude-stijl die de computer een aantal gegevensbestanden laat verwerken.

Bij het beoordelen van JCL's op aspecten die in hogere programmeertalen gemeengoed zijn geworden, vallen direkt een aantal gebreken in het oog: Het formaat is nogal star; vaak wordt een JCL-programma geacht kaartbeelden te omvatten, waarbij de kolom waar een commando begint terzake doet. Variabelen schitteren door afwezigheid, of zijn alleen in rudimentaire vorm aanwezig, b.v. een vast aantal globale variabelen die alleen een karakter kunnen bevatten. Aan besturingstructuren is doorgaans niet veel meer te vinden dan de mogelijkheid een groepje commando's afhankelijk van een bepaalde conditie uit te voeren. Dus geen while-constructie en geen procedures (soms wel macro's). Zoals uit de opsomming blijkt, ontbreken eigenlijk alle eigenschappen die gekoesterd worden door de groeiende schare aanhangers van de methodiek van het gestructureerd programmeren.

Door een JCL te ontwerpen als een hogere programmeertaal, voorzien van de algemeen aanvaarde uitdrukkingsmiddelen en daarnaast een aantal primitiva die het mogelijk maken het bedrijfssysteem te bespelen, zal onmiddelijk een

geweldige vooruitgang worden gemaakt. Het ligt voor de hand dat de verwerking van een programma in zo'n hogere JCL een compilatiefase vergt. De extra kosten die dit met zich brengt hoeven niet zwaar beoordeeld te worden: vooralsnog hebben JCL-programma's niet de neiging zeer groot te zijn, en het aantal aangeboden programma's per dag is gelijk aan: beschikbare rekentijd per dag / gemiddelde rekentijd per job. Aangezien de gemiddelde rekentijd per job minder snel blijkt te dalen dan omgekeerd evenredig met de toenemende rekencapaciteit van de apparatuur, zullen deze extra kosten de neiging hebben te dalen. Bovendien, en dit is een zwaarder wegend argument, moet in de vergelijking ook betrokken worden de extra kosten die nu gemaakt worden doordat de gebruiker zich in bochten moet wringen om zijn bedoeling in JCL uitgedrukt te krijgen, en de kosten veroorzaakt door een foutief JCL-programma dat in een hogere taal uitgedrukt niet fout zou zijn opgeschreven of bij de compilatiefase door de mand zou zijn gevallen. De kosten veroorzaakt door JCL-fouten worden geschat op 3% van de globale rekenkosten; voor het jaar 1975 zou dat neerkomen op $ $1.45_{10}9$ (ENSLOW [3]).

Wel is enige voorzichtigheid geboden, wanneer erop gemikt wordt dat de commando's van de JCL ook interactief gebruikt kunnen worden; de mogelijkheid van interpretatie of stapsgewijze compilatie lijkt dan geboden.

2. WAAROM EEN APARTE JCL?

Is het wel zinvol een aparte JCL te ontwerpen? Waarom niet een bestaande hogere programmeertaal te nemen en die uit te breiden met faciliteiten, zoals bibliotheekprocedures, die toegang geven tot het bedrijfssysteem? BARRON & JACKSON [1] argumenteren dat een bedrijfssysteem een aantal compilers, editors en algemene gebruiksprogramma's ter beschikking moet stellen en dat het job-besturingssysteem verbindingen moet leggen tussen een aantal systemen die verschillende taalconventies hanteren, zodat een systeemgerichte taal voor job-besturing al even noodzakelijk lijkt als een probleemgerichte taal voor het programmeren. LAUESEN [4] bepleit juist het omgekeerde, al kan men, formalistisch redenerend, volhouden dat de door hem voorgestelde taaluitbreidingen in feite een nieuwe, nu (ook) systeemgerichte taal creëren. Voorbeelden van deze aanpak zijn ABLE van PARSONS [5], een op EULER gebaseerde JCL (zie ook RAYNER [6]) en een door JENSEN & LAUESEN [7] voorgestelde uitbreiding van ALGOL 60. De argumenten voor en tegen vergelijkend, komen GRAM & HERTWECK [8] tot de slotsom dat de argumenten tegen de inbouw van

JCL-elementen in bestaande hogere programmeertalen zwaarder lijken te wegen dan de argumenten voor. Niet alle aangevoerde argumenten zijn bij nadere beschouwing even sterk; met name het argument dat programmeertaal en JCL "hoogstwaarschijnlijk" niet samenvallen omdat de een probleem- en de ander systeemgericht is gaat geheel voorbij aan de door Lauesen ingebrachte gedachten, en heeft meer weg van het uitgaan van hetgeen te bewijzen was. Zoals HERTWECK [9] zelf opmerkt, zijn de huidige "probleemgerichte" talen in werkelijkheid "programmeurgericht", met andere woorden, gericht op de gebruikers van die talen. Maar dan moet een goede JCL ook gericht zijn op de gebruiker i.p.v. op het systeem, zodat de bodem uit het argument valt.

Veel belangwekkender argumenten worden aangevoerd door RAYNER [6]. De inbouwmethode à la Lauesen blijft in wezen beperkt tot een machine-afhankelijke (of eigenlijk bedrijfssysteem-afhankelijke) oplossing. Om de analogie van Barron & Jackson te gebruiken: een assembler, ingebed in een hogere taal. Maar de essentie van hogere programmeertalen is juist dat de gebruiker zijn bedoelingen kan uitdrukken in zijn eigen termen, zonder bij voorbeeld het instructie-repertoire van de doelmachine te kennen. Zo moet ook de gebruiker van een JCL zich kunnen uitdrukken, zonder daarbij te hoeven refereren aan de specifieke riten die door een bepaald systeem worden gevergd. Zo'n hogere JCL is dan te vergelijken met een hogere programmeertaal als FORTRAN of ALGOL 68. De implementator heeft te zorgen dat de semantiek van de JCL op de een of andere wijze wordt afgebeeld op de faciliteiten van het bedrijfssysteem - hoe hij dat doet is zijn zaak. Een opmerking als "dat kun je niet in JCL uitdrukken, want het bedrijfssysteem heeft niet zo'n faciliteit" hoort even dwaas te klinken als "op onze computer kun je geen FORTRAN draaien, want hij heeft geen floating-point instructies". Bij de huidige stand van zaken is het nog te vroeg om aan de standaardisering van een hogere JCL te beginnen; we moeten nog de geschikte structuren vinden om job-besturingsproblemen in uit te drukken en de wezenlijke concepten bepalen die een kern van faciliteiten geven waarop kan worden voortgebouwd.

De invloed van hogere programmeertalen op de architectuur van computers is vrij gering (TANENBAUM [10]). Bij JCL's ligt de situatie gunstiger: bedrijfssystemen zijn tenslotte software en kunnen gemakkelijk ontwikkeld worden om aan bepaalde eisen te voldoen. Zo zou de invoering van gestandaardiseerde, systeem-onafhankelijke JCL's de basis kunnen leggen voor toekomstige ontwerpen van bedrijfssystemen. Dit laat de wens onverlet, dat de faciliteiten van het bedrijfssysteem, waar mogelijk, in hogere programmeertalen ter beschikking worden gesteld. Ook is de mogelijkheid niet bij voor-

baat uitgesloten dat er talen uit de bus komen die zeer wel een plaats weten te vinden naast bestaande hogere programmeertalen.

3. HET PROFIEL VAN EEN HOGERE JCL

Wie zich aan de taak zet een systeem-onafhankelijke, hogere JCL te ontwerpen, zal een idee moeten hebben van de basisconcepten, de semantische primitiva waarop de taal moet worden gefundeerd. De aanpak die dan voor de hand ligt is een overzicht te maken van concepten die min of meer gemeenschappelijk zijn aan een grote verscheidenheid van bedrijfssystemen en die dan in de te ontwerpen JCL moeten zijn uit te drukke. Op deze wijze wordt een profiel opgebouwd, een lijst van specificaties voor de gewenste hogere JCL, ongeveer zoals bij sommige sollicitatieprocedures een profiel wordt samengesteld van de ideale kandidaat.

Wanneer we deze oefening doortrekken naar de analogie met assembleer- versus machine-onafhankelijke hogere programmeertalen, dan zou het profiel dat uit deze bezigheid voortvloeit voor de hogere taal wel eens als volgt eruit kunnen komen te zien:

- er moeten accumulatoren zijn waarin tussenresultaten van berekeningen kunnen worden opgeslagen;
- de accumulatoren kunnen worden geladen met de inhoud van een geheugenelement, en de inhoud van een accumulator kan daar worden opgeborgen;
- afhankelijk van de inhoud van een accumulator moet de berekening zijn normale gang kunnen onderbreken en elders worden voortgezet;

enzovoorts.

Het is duidelijk: wat op deze wijze wordt verkregen is niet het profiel van een hogere programmeertaal, maar van een machine-onafhankelijke assembleertaal. Het ontwerpen van deze taal komt neer op het specificeren van een machine-onafhankelijke machine. Op dezelfde wijze is te verwachten dat bovenomschreven oefening uitmondt in het specificeren van een systeem-onafhankelijk bedrijfssysteem. En dan waarschijnlijk een bedrijfssysteem dat een soort grootste gemene deler is van de systemen uit de "hoofdstroming", waarbij de vaak fundamentele inbreng van geïsoleerde, experimentele systemen niet kan worden meegenomen.

Daarom is het noodzakelijk terug te gaan naar de vraag: wat wil de gebruiker, die zijn job ter verwerking aanbiedt, tot uitdrukking brengen? Een moeilijk te beantwoorden vraag, aangezien de huidige gebruikers gecon-

ditioneerd zijn door de systemen die ze gebruiken, en hun antwoorden alleen kunnen formuleren in de begrippen die ze via die systemen hebben leren kennen. Een richting naar een uitweg uit deze impasse wordt gesuggereerd door te kijken naar de kenmerkende verschillen tussen de opgave waarvoor een programmeur staat die een "probleem" moet oplossen, en de gebruiker die de besturing van een job moet beschrijven.

In beide gevallen moet een proces worden gespecificeerd, gebruik makend van de uitdrukkingsmogelijkheden van een taal. Voor de programmeur is hierbij kenmerkend dat hij als het ware met een schone lei begint, en kan kiezen uit een arsenaal van goed gedefinieerde, zeer algemene primitiva. Voor zover het te beschrijven proces wordt opgebouwd uit deelprocessen, kan hij in het algemeen zelf kiezen hoe die structurering wordt opgelegd, en volgens welke conventies de koppeling plaatsvindt. Zo bepaalt hij zelf type en aantal parameters van de procedures die hij schrijft, en de reactie op bijzondere situaties die de procedure niet aankan. Geheel anders ligt de situatie bij het specificeren van ee job-besturingsproces. Daar zijn juist in het algemeen een aantal kant-en-klare deelprocessen gegeven, met hun al bepaalde conventies. De opgave is nu op de een of andere wijze deze processen aan elkaar te knopen, te zorgen dat een goede koppeling tot stand komt. Ook de reactie op onvoorziene omstandigheden is vooruit bepaald. Een hogere JCL zal in de eerste plaats voorzieningen moeten hebben om aan deze omstandigheden het hoofd te bieden. Hiervoor is een bepaalde, liefst zo uniform mogelijke benadering van proceskoppeling gewenst.

In de rest van dit verhaal wordt een bepaalde benadering gekozen, en wordt onderzocht wat daarmee kan worden uitgedrukt. Daarbij is uitdrukkelijk niet uitgegaan van de "behoeften" van een bedrijfssysteem, en belangrijke vragen als: hoe is de toewijzing van schaarse faciliteiten geregeld? of: hoe lost het systeem "deadlock" op? blijven onaangeraakt. Ook is uitdrukkelijk niet nagegaan in hoeverre een efficiënte implementatie mogelijk is op bestaande of nog te ontwikkelen bedrijfssystemen. Bij de huidige stand van zaken zou dat alleen maar een blok aan het been betekenen bij pogingen door te dringen tot de kern van de zaak.

4. STROMEN

Centraal in de benadering die hier gekozen is, staat het begrip "stroom". Dit begrip vertoont een zekere verwantschap met de "stream" van

JENSEN & LAUESEN [7], die weer een soort generalisatie is van de sequentiële file. Een andere nauwe verwant is de "pipe" uit het UNIX-systeem (RITCHIE & THOMPSON [11]).

Een eenvoudige beschrijving is te geven door een stroom voor te stellen als een queue: een sequentieel magazijn met een ingang en een uitgang. De queue bevat "elementen"; daarvoor zullen we in het vervolg steeds karakters nemen, maar het verdient aanbeveling ook algemener mogelijkheden voor ogen te houden. Bij afspraak hebben alle nieuw geschapen stromen een lege queue.

Processen kunnen karakters naar een stroom sturen of van een stroom halen. Stromen kunnen dus gebruikt worden als communicatiemiddel tussen processen. De operaties om een karakter c naar of van een stroom s te sturen resp. ontvangen, worden genoteerd als *put (s, c)* resp. *c:= get (s)*. Een speciaal karakter *eos* (end of stream) wordt gebruikt om een stroom af te sluiten.

Het eenvoudigste proces dat op twee stromen kan worden toegepast is de "hevel": het domweg doorverbinden van de twee stromen. Notatie: $a \rightarrow b$, met betekenis

```
while char c = get (a);
      c ≠ eos
do put (b, c) od.
```

Hierbij vallen een paar kanttekeningen te maken. Het is denkbaar dat de queue van stroom a leeg is op het moment van de aanvraag *get (a)*, of dat de queue van b over zou stromen bij het aanbod *put (b, c)*. De bedoeling is dat in zo'n geval het proces $a \rightarrow b$ wordt opgeschort tot het weer ongehinderd kan voortgaan. Een gevolg is dat het overhevelingsproces niet autonoom hoeft te lopen, maar in feite gestuurd kan worden door hetzij het aanbod via a, hetzij de aanvragen via b.

Om met een string x een stroom s_x te laten corresponderen die op aanvraag de achtereenvolgende karakters van x aflevert, kunnen we schrijven:

```
begin  for i to upb x
       do  put  (s_x, x[i]) od;
       put (s_x, eos)
end.
```

We spreken af dat het optreden van een string x op een positie waar een stroom hoort steeds deze corresponderende stroom s_x zal beduiden. Dan is de betekenis van $x \rightarrow s$ dezelfde als van

```
par (# x = s_x # begin for i to upb x
                       do put (s_x, x[i]) od;
                       put (s_x, eos)
                   end,
     # s_x → s # while char c = get (s_x);
                       c ≠ eos
                   do put (s, c) od).
```

Het is niet moeilijk in te zien dat dit vereenvoudigd kan worden tot

```
for i to upb x
do put (s, x[i]) od.
```

In feite kunnen we dus de betekenis van x als stroom op deze wijze definiëren:

```
x → s ⇐ for i to upb x
        do put (s, x[i]) od.
```

Gebruikmakend van deze definitiewijze kunnen we een operator + beschrijven om twee stromen aan elkaar te knopen:

```
a + b → s ⇐ (a → s; b → s).
```

We zien hier dat $a + b$ met een stroom correspondeert die zijn karakters uit a haalt totdat a is uitgeput, waarop op b wordt overgeschakeld.

Tot slot van deze paragraaf een wellicht niet zo zinvolle, maar wel illustratieve mogelijkheid om met een stroom te manipuleren: een operator *odd*, die uit een stroom het eerste, derde, vijfde enz. karakter kiest:

```
odd a → s ⇐ repeat char c:= get (a);
                if c ≠ eos
                then put (s, c);
                     c:= get (a)
                fi
            until c = eos.
```

5. PROCESSEN

Om een uniforme behandeling te krijgen, nemen we aan dat alle processen werken op stromen. De koppeling van processen vindt dan plaats door ze via een stroom te laten communiceren. Voorzover dit, door verschil in conventies, niet rechtstreeks kan, moet een "transformator" worden tussengeschakeld.

Laat *pro* de naam van een proces zijn, dat zijn gegevens van een stroom *in* haalt en zijn resultaten in een stroom *out* aflevert. Wanneer dit proces gebruikt moet worden om op een stroom *data* te werken en zijn resultaten in de stroom *print* af te leveren, kan dat proces beschreven worden door parallel twee processen naast *pro* op te zetten, één dat *data* overhevelt naar *in*, en één dat *out* naar *print* overhevelt:

pro, data → in, out → print.

In een globale omgeving zou een proces kunnen bestaan, dat standaard is opgezet als *print → print device*. Wie dat niet wenst, kan in plaats daarvan het proces *print → prod 23* opzetten, zodat de print-stroom naar elders wordt gedirigeerd. Hierbij kan de "aanroep" van *pro* onveranderd blijven. Andere mogelijke aanroepen worden gegeven door

pro, "3.14, 2.72" → in, out → print

en

pro, data 1 + data 2 + "3.14" → in, "trial" + nr + out → print

Wanneer *pro 1* een proces is dat gegevens krijgt van een stroom *a* en resultaten aflevert in *b*, terwijl *pro 2* resultaten krijgt van *b* en aflevert in *c*, dan wordt een direkte koppeling tot stand gebracht door ze parallel te zetten:

pro 1, pro 2.

Een proces kan op veel meer dan twee stromen opereren, zodat ook aanroepen denkbaar zijn als

a68, prog → source, list → print, err → print, read → standin, standout → print.

Wanneer in de omgeving al een stroom *source* bestaat die als invoer moet

worden genomen, en het proces *list → print* al is opgezet, kan volstaan worden met

a68, err → print, read → standin, standout → print.

Nog korter:

a68, read → standin, err + standout → print.

Deze laatste schrijfwijze garandeert dat wat door *a68* in *err* wordt afgeleverd, vóór *standout* in *print* terechtkomt, terwijl in het eerste geval, wanneer we aannemen dat de overheveling naar *print* willekeurig snel gebeurt, de volgorde die is waarin de karakters door *a68* worden afgeleverd.

6. LOCALITEIT

Een kenmerk van hogere programmeertalen is het localiteitsprincipe: identifiers worden in een bepaalde context geïdentificeerd, waarbij een blokstructuur van het programma er garant voor staat dat beschrijvingen van deelprocessen niet onnodig interfereren met elkaar of met de beschrijving van een globaler proces. Van een hogere JCL verwachten we hetzelfde.

De eerder geïntroduceerde schrijfwijze vertoont een treffende overeenkomst, zowel uiterlijk als qua effect, met benoemde parameters, als in een conventionaal JCL-commando

PRO, IN = DATA, OUT = PRINT.

Het zou niet wenselijk zijn wanneer de *in* en *out* "parameters" van *pro* telkens locaal gedeclareerd zouden moeten worden, als in

begin stream in, out;
pro, data → in, out → print
end.

Overigens is het wel de bedoeling dat dit mogelijk is, dus dat de stroomidentifiers die bij een bepaald proces horen voor een bepaalde aanroep vanuit die positie geïdentificeerd worden, in plaats van vanuit de positie van de definitie. Maar dan nog zou de getergde gebruiker er al gauw toe overgaan op (te) globaal niveau eens en vooral de meest gebruikte stroom-

identifiers te declareren. Daarom geldt de volgende afspraak: Een proces-identifier bevat impliciete voorkomens van de gebruikte stromen (dus *pro* bevat impliciet *in* en *out*. Een stroom-identifier die in een lijst parallele processen impliciet voorkomt in een proces-identifier, en ook (impliciet of expliciet) daarbuiten, wordt geacht impliciet gedeclareerd te zijn in een blok dat de lijst onmiddelijk omgeeft. Volgens deze regel is dus

pro 1, pro 2

equivalent met

begin stream b; pro 1, pro 2 end.

Een wat uitgebreider voorbeeld, waarbij de impliciete voorkomens tussen accolades vermeld zijn:

compile {source, obj, list}, prog → source, list → formatter,
load {obj, entry}, run {entry, data, out}, out → print.

Dit geheel bevat impliciete declaraties van de stromen *source, obj, list, entry* en *out*, zodat alleen *data* niet locaal gedeclareerd is. In hoeverre deze conventie voldoende verrassingsvrij is valt nog te bestuderen.

Een sterker voorbeeld van localiteit kan worden gegeven door anonieme stromen. Een proces met één in- en één uitvoerstroom kan beschouwd worden als een transformator. Voorbeelden van processen die conceptueel in de eerste plaats transformatoren zijn, zijn het weghalen of toevoegen van "control characters" voor een regeldrukker, het terugbrengen naar een regelbreedte van b.v. 80 karakters of code-conversie. Laat zo'n proces de naam *trafo* hebben. We definiëren nu

a @ trafo → s ⇐ trafo, a → in, out → s.

(De eerder beschreven operator *odd* kan ook als transformator gedefinieerd worden.) Willen we de transformatie tweemaal in successie toepassen, dan kunnen we ook schrijven

a @ trafo @ trafo → s.

7. EDITING

Een conventionele wijze van editing wordt gesuggereerd door een aanroep die een in- en outvoerstroom kunt en de aan te brengen veranderingen:

edit, text → in, "/JCL/=/Job Control Language/" → changes, out → text 1.

Het is echter ook mogelijk een edit-commando als een transformatie te beschouwen. We zouden dan kunnen schrijven:

text @ /JCL/=/Job Control Language/ → text 1.

Bij deze laatste schrijfwijze wordt de mogelijkheid geopend edit-transformaties en andere, b.v. zelf gedefinieerde, transformaties door elkaar te gebruiken.

In het algemeen heeft een edit-operatie de volgende gedaante: een gedeelte dat een procédé bevat om een stuk tekst te localiseren en een gedeelte dat daarvoor een vervanging verschaft. Wanneer dat procédé iets algemener gekozen wordt dan gebruikelijk is, bijvoorbeeld een regulier patroon met context, wordt de machtigheid aanzienlijk uitgebreid. Een wezenlijke verbetering wordt verkregen als de vervanging zelf wordt gezien als een transformatie, dus als een proces dat invoer krijgt (de tekst die op het patroon bleek te passen) en uitvoer (de nieuwe tekst) aflevert. Bij deze zienswijze is b.v. */Job Control Language/* een zodanige transformatie, dat

x @ /Job Control Language/

equivalent is met *"Job Control Language"* (als stroom). Wanneer een transformatie naast zijn hoofd in-en-uitvoerstroom ook nevenstromen kan gebruiken, is het denkbaar dat een stroom het rgelnummer van voorkomen levert, en dat op een andere stroom zo alle regelnummers worden gezet waar een bepaalde identifier voorkomt. Een andere mogelijkheid is via stromen stukken tekst te rangeren, zodat bijvoorbeeld formulas in reverse Polish (postfix) notatie worden omgezet.

Een stuk uit te voeren JCL kan zelf door editing en dergelijke ontstaan zijn. Aangezien de JCL-compiler zelf een onderdeel zal zijn van het systeem, moet het mogelijk zijn dit als stroom toe te voeren en uitgevoerd te krijgen, door een aanroep als

jcl, s → in.

Het is beter hiervoor een aparte notatie in te voeren, zoals

: s.

In combinatie met editing geeft dit een macro-faciliteit.
Wanneer b.v. een string *run* is gedefinieerd door

*string run = "a68, * → source, list → * list",*

dan is

: run @ //=/myprog/*

equivalent met

a68, myprog → source, list → myproglist.

8. SYNCHRONISATIE EN FOUTBEHANDELING

Bij een proces als $a \rightarrow c$, $b \rightarrow c$ is het niet gedefinieerd in welke volgorde de karakters uit de stromen a en b naar c worden overgeheveld. Toch kan het wenselijk zijn dat twee processen die b.v. van dezelfde stroom lezen of ernaar schrijven gesynchroniseerd verlopen: steeds krijgt een proces de beurt tot het bereid is die af te staan.

De bekendste synchronisatiemethode is die van de semaforen. Wanneer een proces een semafoor tracht neer te halen die al neer is, wordt het opgeschort; het kan pas door wanneer een ander proces de semafoor weer omhoog zet.

Dit lijkt als twee druppels water op de situatie waarin een proces een karakter van een stroom probeert te halen die leeg is (maar niet afgesloten: dan is er iets fout): het proces wordt opgeschort en moet wachten tot een ander proces iets in de stroom zet. Hieruit volgt dus dat alles wat met semaforen gedaan kan worden, ook met stromen kan. Stromen geven in zoverre meer flexibiliteit dat ook nog informatie wordt gegeven door datgene wat in de stroom is geschreven.

In de vorige alinea is (voor het eerst) gezinspeeld op de mogelijkheid dat er iets fout kan gaan. Het is noodzakelijk dat een job-besturend programma daarmee rekening kan houden. Niet alleen vanwege hardware-fouten, maar ook vanwege de mogelijke onbetrouwbaarheid van processen. Tenslotte heeft de gebruiker niet in de hand dat een hem ter beschikking gesteld programma alle situaties aankan die het zou moeten aankunnen. Hoe kan nu in JCL gespecificeerd worden hoe op fatale fouten gereageerd moet worden?

Voor een oplossing kunnen we teruggrijpen naar de tijd dat de operateur nog een job-besturende rol had. Zijn rol was een proces, parallel met de verwerkingsprocessen. In die hoedanigheid kon hij kennis nemen van fatale fouten, ook wanneer daarvoor het verkeringsproces beëindigd was. Analoog daaraan is het mogelijk parallel aan willekeurig welk proces een "operateursproces" op te zetten. Als de afspraak wordt gemaakt dat fatale fouten altijd resulteren in een melding op een stroom met de standaardnaam *fatal*, hoeft het operateursproces niets anders te doen dan te wachten op zo'n melding en naar bevind van zaken te reageren. Is op dat niveau niet zo'n proces aanwezig, dan wordt de melding gedirigeerd naar de stroom *fatal* op een hoger niveau (in de hoop dat daar wel een operateur is). Deze oplossing lijkt op die van het "complaint-department" in PLASMA (GREIF & HEWITT [12]).

9. CONCLUSIE

In de voorafgaande paragrafen is geprobeerd aannemelijk te maken dat het begrip "stroom" voldoende mogelijkheden biedt om op uniforme wijze proceskoppelingen te beschrijven, en daardoor kan dienen als een van de semantische kernelementen waarop hogere JCL kan worden opgebouwd. Hier nog enkele woorden over een mogelijke implementatie: Er bestaat geen noodzaak, voor de stroom een uniforme interne representatie te kiezen. Het is waarschijnlijk veel beter van geval tot geval die keuze te laten afhangen van het gebruik dat van de stroom wordt gemaakt. Door een routine voor de operaties get en put als onderdeel van iedere representatie op te nemen (REYNOLDS [13], STOY & STRACHEY [14]) kan via een conventie die extern uniform is toch de noodzakelijke koppeling worden verzorgd. Ook hoeft voor een koppeling als $a \rightarrow b$ niet per se een proces te worden opgezet; in veel gevallen, die statisch kunnen worden ontdekt, kan dit beschouwd worden als een symbolische equivalentie-aanduiding, zodat de verbinding kan worden kortgesloten.

Geen aandacht is besteed aan het gebruik dat gemaakt kan worden door het overnemen van elementen uit hogere programmeertalen in een JCL. Het zal vanzelf spreken dat er situaties zijn waar met vrucht gebruik kan worden gemaakt van b.v. procedures of een while-constructie. Voor het begrip "stroom" is van direct belang de "types" of "modes" van hogere programmeertalen. Als een stroom een mode heeft die afhangt van de mode van de af te leveren elementen, wordt een redundantie bereikt waardoor statisch veel fouten kunnen worden voorkomen. Dan wordt het bij voorbaat ontdekt, wanneer de gebruiker bij ongeluk een niet vertaalde brontekst als uitvoerbare code wil doen laden, of een binair bestand met eën daarop niet berekende editor te lijf wil. Al dan niet impliciete standaardconversies, die de representatie onveranderd kunnen laten, zorgen voor de gewenste flexibiliteit.

LITERATUUR

[1] BARRON, D.W. & I.R. JACKSON, *The evolution of Job Control Languages*, Software - Practice & Experience 2 (1972) 143-164.

[2] UNGER, C. (ed.), *Command Languages*, Proc. IFIP Working Conference on Command Languages, North-Holland, Amsterdam, 1975.

[3] ENSLOW Jr., P.H., *Summary of the IFIP Working Conference on Operating System Command Languages*, in [2], 389-394.

[4] LAUESEN, S., *Program Control of Operating Systems*, BIT 13 (1973) 323-337.

[5] PARSONS, I.T., *A high-level Job Control Language*, Software - Practice & Experience 5 (1975) 69-82.

[6] RAYNER, D., *Recent Developments in Machine-Independent Job Control Languages*, Software - Practice & Experience 5 (1975) 375-393.

[7] JENSEN, J. & S. LAUESEN, *Programming Language extensions which render Job Control Languages superfluous*, in [2], 137-151.

[8] GRAM, C. & F.R. HERTWECK, *Command Languages: design considerations and basic concepts*, in [2], 43-67.

[9] HERTWECK, F.R., opmerking in *Panel Discussion A*, in [2], 209-220.

[10] TANENBAUM, A.S., *Programming Languages and Hardware*, Colloquium Structuur van Programmeertalen, MCS 25, Mathematisch Centrum, 1976.

[11] RITCHIE, D.M. & K. THOMPSON, *The UNIX Time-Sharing System*, Comm. of the ACM 17 (1974) 365-375.

[12] GREIF, I. & C. HEWITT, *Actor Semantics of Planner-73*, Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, January 20-22, 1975.

[13] REYNOLDS, J.C., *User-defined types and Procedural data structures as complementary approaches to data abstraction*, in New Directions in Algorithmic Languages, 1975, 154-165, S.A. Schuman (ed.), IRIA, 1976.

[14] STOY, J.E. & C. STRACHEY, OS-6, *An experimental operating system for a small computer; Part 2: Input/output and filing system*, The Computer Journal 15 (1972) 195-203.

MATHEMATICAL CENTRE SYLLABUS 32, 1976, VIII, 129-146

OVER DE IMPLEMENTATIE VAN SYNCHRONISATIECONCEPTEN

H.J.M. GOEMAN
Rijksuniversiteit, Leiden

0. INLEIDING

Bij het gelijktijdig plaatsvinden van verschillende (sequentiële) processen *) in een gemeenschappelijke omgeving is het dikwijls wenselijk of noodzakelijk de gedragingen van afzonderlijke processen te coördineren. Het opleggen van beperkingen aan het onafhankelijk funktioneren van afzonderlijke (sequentiële) processen wordt aangeduid met de term synchronisatie van processen.

Synchronisatie is van belang in de volgende situaties:

- wanneer het gewenste cumulatieve effect van de verrichtingen van verschillende (sequentiële) processen bij gelijktijdig plaatsvinden van die verrichtingen niet meer gewaarborgd is (vb.: het gelijktijdig wijzigen van de toestand van een gemeenschappelijke informatiedrager in de gemeenschappelijke omgeving);
- wanneer verrichtingen van verschillende (sequentiële) processen een bepaalde volgorde in de tijd vereisen (vb.: signalen tussen processen moeten worden verzonden voordat ze kunnen worden ontvangen);
- indien schaarste aan bepaalde hulpmiddelen die voor exclusief gebruik gewenst zijn daartoe noodzaakt (vb.: het gebruik van buffers, het verwerven van geheugen voor eigen gebruik).

---

*) Onder een proces verstaan we het geheel van akties dat als gevolg van één enkele individuele uitvoering van een programma in een zekere omgeving plaats vindt. Als het geheel van akties bestaat uit een (eindige) reeks van opeenvolgende akties dan spreken we van een sequentieel proces. Uitvoering van een sequentieel programma resulteert steeds in een sequentieel proces.

1. DE KRITIEKE SECTIE

Een van de belangrijkste synchronisatieconcepten is het concept van de kritieke sectie. We stellen ons een aantal (zeg N) sequentiële programma's voor die gelijktijdig worden uitgevoerd. Van bepaalde programmastukken, de kritieke secties genoemd, wordt verlangd dat op ieder moment hoogstens één van de resulterende processen aan de uitvoering van zo'n kritieke sectie bezig mag zijn. We kunnen de levensloop van het j-de proces dus beschrijven als een afwisseling van kritieke en niet-kritieke secties:

```
process(j): while ¬ finished(j) do
                critical section;
                noncritical section
            od
```

Nu is het heel voor de hand liggend om in de gebruikte programmeertaal een syntaktische konstruktie op te nemen waarmee programmastukken tot kritieke secties kunnen worden verklaard en om vervolgens aan uitvoeringen van zulke programma's de eis te verbinden dat uitvoeringen van kritieke secties elkaar wederzijds in tijd dienen uit te sluiten.

```
process(j): while ¬ finished(j) do
                enter
                     critical section
                leave;
                noncritical section
            od
```

Aldus hebben we, de implementeerbaarheid veronderstellend, een uitstekend instrument in handen gekregen om nadere kontrole op de synchronisatie van gelijktijdig verlopende processen uit te oefenen. Maar de vraag naar de implementeerbaarheid van de kritieke sectie is daarmede natuurlijk niet opgelost. Het probleem van de implementeerbaarheid vraagt immers naar de middelen die kunnen worden aangewend om de wederzijdse uitsluiting van de uitvoering van kritieke secties daadwerkelijk te realiseren. Het vraagt bovendien naar het ritueel dat de processen bij het binnengaan en het verlaten van de kritieke sectie moeten volgen om te verzekeren dat steeds ten hoogste één van de processen in een kritieke sectie zal verkeren. Met andere woorden, het vraagt naar de programmering van enter en leave in

```
process(j) : while ¬ finished(j) do
                    enter;
                    critical section;
                    leave;
                    noncritical section
             od
```

Voor het geval van twee processen werd dit probleem in het begin van de zestiger jaren (vermoedelijk in 1962) opgelost door Th. J. DEKKER en later voor het algemene geval van N processen door E.W. DIJKSTRA [1]. Dijkstra's oplossing werd vervolgens nog geamendeerd door KNUTH [2] en door EISENBERG & McGUIRE [3]. Alle genoemde oplossingen gingen uit van de veronderstelling dat inspecties van de waarde van een gemeenschappelijk informatiedrager of toekenning van een nieuwe waarde aan een gemeenschappelijke informatiedrager kunnen worden beschouwd als ondeelbare operaties die elkaar in tijd uitsluiten. Met andere woorden: indien twee of meer processen gelijktijdig een nieuwe waarde zouden willen toekennen aan dezelfde gemeenschappelijk toegankelijke informatiedrager dan zullen die toekenningen in één of andere volgorde één voor één plaats vinden en de uiteindelijke waarde zal bepaald zijn door de toekenning die het laatst plaats vindt. Zo zal, indien gelijktijding een inspectie van en een toekenning aan een gemeenschappelijke informatiedrager plaats vinden, de inspectie ofwel de oude waarde voorafgaande aan de toekenning ofwel de nieuwe waarde na de toekenning maar nooit een of ander mengsel van beiden kunnen opleveren.
We zullen deze veronderstelling in het vervolg aanduiden als het postulaat van de atomiciteit van inspecties en toekenningen aan gemeenschappelijke informatiefragers.

DIJKSTRA publiceerde zijn oplossing in 1965 en somde daarbij een aantal kriteria op waaraan zijn oplossing voldeed.
Wij formuleren die kriteria als volgt:

1) op ieder moment zal ten hoogste één van de processen zich in zijn kritieke sectie bevinden;
2) de oplossing mag geen enkele veronderstelling bevatten over de relatieve snelheden van de verschillende processen;
3) de oplossing is in die zin symmetrisch ten opzichte van de processen dat zij geen statische prioriteitsregels mag bevatten;
4) indien één of meer processen (tijdelijk) niet meer wil deelnemen aan de

competitie om de kritieke secties te betreden en zich in een niet kritieke sectie ophoudt (of daar zelfs stopt) dan zal dat geen gevolgen mogen hebben voor de mogelijkheden van ieder van de overige processen om de kritieke sectie te kunnen betreden;

5) indien geen van de processen zich in zijn kritieke sectie bevindt en één of meer processen de competitie aangaan om de kritieke sectie te betreden dan zal gegarandeerd binnen eindige tijd één van de processen de kritieke sectie binnengaan.

De oplossingen van KNUTH en die van EISENBERG & McGUIRE voegden daaraan respektievelijk nog de volgende kriteria toe:

6) indien ieder van de processen zich telkens hoogstens een eindige tijd in de kritieke sectie ophoudt, dan is het onmogelijk om zodanige relatieve (positieve) snelheden voor de processen aan te geven dat de toegang voor enig proces door toedoen van de overigen voortdurend effectief geblokkeerd blijft;

7) zodra een van de processen het ritueel aanvangt om de kritieke sectie binnen te gaan dan zal dat proces na hoogstens N-1 beurten waarin andere processen nog kunnen voorgaan de kritieke sectie betreden.

Dekker's oplossing voor twee processen voldoet aan de kriteria 1 t/m 6. We geven hier een naar de vorm lichtelijk vereenvoudigde versie van zijn oorspronkelijke oplossing. Het programma voor process(j) met j=1,2 luidt:

```
process(j) : while ¬ finished(j) do
                 interest(j) := true;
                 while interest(3-j) do
                       while turn = 3-j do interest(j) := false od;
                       interest(j) := true
                 od;
                 critical section;
                 turn := 3-j; interest(j) := false;
                 noncritical section
             od
```

Hierin is turn een integer, geïnitialiseerd met de waarde 1 of 2, interest(j) is evenals finished(j) een boolean geïnitialiseerd met de waarde false voor j=1,2.

De oplossing is veilig (krit.1) want process(j) zal alleen dan de kritieke

sectie binnengaan als het zijn eigen geïnteresseerdheid daartoe heeft aangegeven (met interest(j) := true) en vervolgens heeft geconstateerd dat de ander niet geïnteresseerd is (interest(3-j) = false). Process(j) zal bovendien zijn geïnteresseerdheid pas weer opgeven nadat het de kritieke sectie heeft verlaten zodat de toegang voor het andere proces in ieder geval tot dat moment geblokkeerd zal blijven.
Indien een van beide processen (enige tijd) geen belangstelling meer heeft voor de kritieke sectie, dan zal interest(j) voor dat proces steeds de waarde false behouden, waardoor het andere proces ongehinderd en herhaaldelijk de kritieke sectie kan betreden (krit. 4).
Indien beide processen met interest(j) := true hun geïnteresseerdheid hebben aangegeven en tot de ontdekking komen dat ook de ander al geïnteresseerd is, dan zal de waarde van turn één van beiden ertoe brengen om de ander voor te laten gaan (krit. 5).
Als process(j) niet als laatste de kritieke sectie heeft verlaten, zodat turn = j, dan zal het aangeven van de geïnteresseerdheid van process(j) voldoende zijn om te verhinderen dat de ander (eventueel) na het verlaten van de kritieke sectie nogmaals zal kunnen binnengaan (krit. 6).

De oplossingen van DIJKSTRA, KNUTH en EISENBERG & McGUIRE voor het algemene geval van N processen zijn buitengewoon subtiel en ingewikkeld. We zullen ze hier niet behandelen. We vragen de lezer wel om zelf na te gaan waarom Dekker's oplossing niet aan kriterium 7 voldoet en dat de volgende oplossing voor twee processen wel aan alle zeven kriteria beantwoordt.
Voor die oplossing geldt dat claim(j) een boolean is geïnitialiseerd met de waarde false, verder gelden dezelfde initialisaties als in de voorafgaande oplossing.

```
process(j) : while ¬ finished(j) do
                 interest(j) := true;
                 while ¬ claim(j) do
                   if interest(3-j) then while turn = 3-j do od fi;
                   claim(j) := true; if claim(3-j) then claim(j) := false fi
                 od;
                 critical section;
                 turn := 3-j; interest(j) := false; claim(j) := false;
                 noncritical section
             od
```

2. DE ACTIEVE ARBITER

Alle genoemde oplossingen maken uitsluitend gebruik van passieve systeemcomponenten om te bepalen welk van de rivaliserende processen de kritieke sectie mag binnengaan. In de behandelde oplossingen speelt de informatiedrager turn daarbij een heel speciale rol. Een aandachtige beschouwing van de rol van die informatiedrager leidt tot de konklusie dat de oplossingen alleen op de lange termijn symmetrisch zijn, terwijl daarentegen op ieder moment een zekere tijdelijk van kracht zijnde prioriteit aanwezig is om zonodig een beslissing af te dwingen die bij volkomen symmetrisch identiek gedrag van de afzonderlijke processen anders niet tot stand dreigt te komen.
Alle bekende oplossingen kennen een dergelijke passieve scheidsrechter.
En inderdaad lijkt het een logische onmogelijkheid om zonder enige asymmetrie in de situatie het onderscheid te kunnen maken dat nodig is om uiteindelijk slechts een van de zich identiek gedragende processen tot de kritieke sectie toe te laten. Dat doel is immers op zich zelf al een asymmetrische situatie.
Zodra we de scheidsrechter als een noodzakelijke bouwsteen voor de implementatie van de kritieke sectie en ook van andere synchronisatieconcepten herkennen, komt de gedachte aan een aktieve scheidsrechter als een voor de hand liggende mogelijkheid naar voren.
Een aktieve scheidsrechter moet dan worden opgevat als een gespecialiseerde systeemcomponent waarvan de werking kan worden beschreven als een cyclisch sequentieel proces dat permanent en gelijktijdig met de normale processen aktief is.

Uitgaande van het eerder genoemde postulaat van de atomiciteit van inspecties van en toekenningen aan gemeenschappelijke informatiedragers, presenteren we de volgende oplossing van de implementatie van de kritieke sectie met de hulp van een arbiter voor het algemene geval van N processen.
Er is een array attention met initiële waarde false voor ieder van de array elementen:

```
attention : array 1 .. N of boolean
```

en een informatiedrager

```
leaving : boolean
```

eveneens met initiele waarde false.
De arbiter gedraagt zich als volgt:

```
arbiter : begin i : integer;
                i := 1;
                repeat
                   if attention(i) then
                      attention(i) := false;
                      while ¬ leaving do od;
                      leaving := false
                   fi;
                   i := i + 1; if i > N then i := 1 fi
                forever
          end
```

Het programma voor process(j) voor j =1,...,N luidt:

```
process(j) : while ¬ finished(j) do
                attention(j) := true;
                while attention(j) do od;
                critical section;
                leaving := true;
                noncritical section
             od
```

Het is eenvoudig in te zien dat deze oplossing aan alle zeven genoemde kriteria voldoet. Merk allereerst op dat process(j) alleen dan in zijn kritieke sectie kan zijn, indien i = j en de arbiter in de instruktie while ¬ leaving do od wacht op een teken dat process(j) zijn kritieke sectie heeft verlaten. Omdat er slechts één arbiter is en de waarde van i niet kan veranderen voordat de arbiter inderdaad het verlaten van de kritieke sectie door process(j) heeft geconstateerd, is wederzijdse uitsluiting gegarandeerd. Het is verder van belang om op te merken dat process(j) niet kan worden verhinderd om via de instruktie attention(j) := true de wens te kennen te geven de kritieke sectie te betreden, terwijl het vervolgens nooit meer dan N-1 beurten behoeft te wachten om de kritieke sectie binnen te gaan, aangezien de arbiter ieder van de overige processen precies éénmaal bezoekt alvorens zijn aandacht weer tot process(j) te wenden.
Wellicht ten overvloede zij nog opgemerkt dat de arbiter geen beslag legt op de processor(en) die voor de uitvoering van de normale programma's zorg dragen. De arbiters zijn speciaal geconstrueerde systeemcomponenten met een

gespecialiseerde taak bijvoorbeeld om dienst te doen als hulpmiddel bij het ritueel om de wederzijdse uitsluiting van kritieke secties te verwezenlijken of als hulpmiddel bij de implementatie van enig ander synchronisatieconcept.

Alle tot nu toe besproken oplossingen van het probleem van de implementatie van de kritieke sectie veronderstellen eigenlijk dat de kritieke sectie op een lager niveau al verwezenlijkt is. Het is met name het eerder genoemde postulaat van de atomiciteit van toekenningen aan en inspecties van gemeenschappelijke informatiedragers, dat moet worden beschouwd als een min of meer verborgen aanname dat de kritieke sectie op lager niveau reeds is gerealiseerd. Zowel vanuit praktisch als vanuit theoretisch standpunt is die situatie echter volstrekt onbevredigend. De volgende vragen doemen immers onmiddelijk op: Hoe is de kritieke sectie op dat lagere niveau dan wel geïmplementeerd? Kan de methode die daar gebruikt is niet zonder meer toegepast worden om de implementatie ook op het hogere niveau te verwezenlijken? Is het wellicht mogelijk om op het lagere niveau (met de daar geïmplementeerde kritieke sectie) zodanige hulpmiddelen te bouwen dat de implementatie op het hogere niveau een betrekkelijk eenvoudige zaak wordt?
Ondanks de reeds besproken oplossingen staat het probleem van de theoretische implementeerbaarheid van de kritieke sectie tengevolge van de eerstgenoemde vraag opnieuw ter diskussie. We zullen daarom onze oplossing zodanig amenderen dat het postulaat van de atomiciteit van inspecties en toekenningen niet langer nodig is. In eerste instantie zullen we nog wel een andere (zwakkere) veronderstelling postuleren, later zullen we echter ook die veronderstelling achterwege kunnen laten.
We zullen dan de wederzijdse uitsluiting van kritieke secties voor gelijktijdig in een gemeenschappelijke omgeving plaats vindende sequentiële processen hebben gerealiseerd zonder enige (verborgen) aanname dat de kritieke sectie (op lager niveau) al zou zijn verwezenlijkt.
De volgende oplossing maakt gebruik van de boolean arrays:

```
attention : array 1 .. N of boolean;
answered  : array 1 .. N of boolean
```

met initiële waarde false voor ieder van de array elementen.

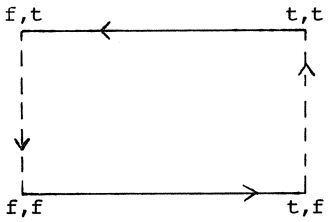
```
arbiter : begin i: integer;
              i := 1;
              repeat
                  if attention(i) then
                     answered(i) := true;
                     while attention(i) do od;
                     answered(i) := false
                  fi;
                  i := i + 1; if i > N then i := 1 fi
              forever
          end
```

Het programma voor process(j), j = 1,...,N, luidt:

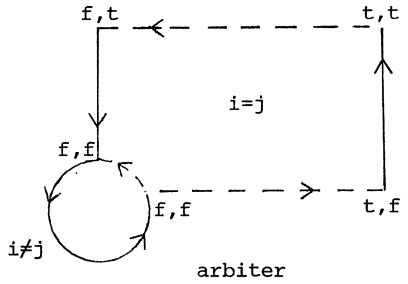
```
process(j) : while ¬ finished(j) do
                 while answered(j) do od;
                 attention(j) := true;
                 while ¬ answered(j) do od;
                 critical section;
                 attention(j) := false;
                 noncritical section
             od
```

Deze oplossing maakt voor de uitwisseling van informatie tussen de arbiter en de N concurerende processen uitsluitend gebruik van tweewaardige (logische) informatiedragers. Bovendien geldt dat ieder van die informatiedragers slechts door één van de processen of alleen door de arbiter zal worden beschreven en ook alleen door de arbiter of door slechts één van de processen zal worden geïnspecteerd. We behoeven dus slechts situaties te beschouwen waarbij toekenning(en) aan een tweewaardige informatiedrager door één schrijvend sequentieel proces zouden kunnen samenvallen met inspectie(s) van één ander lezend sequentieel proces.
Wij behoeven daarom de atomiciteit van toekenningen en instrukties niet te veronderstellen. We kunnen volstaan met een zwakkere aanname:
Wanneer een toekenning en een inspectie gelijktijdig plaats vinden, d.w.z. in niet-disjunkte tijd, dan zal de inspectie ofwel de oude ofwel de nieuw te schrijven waarde mogen opleveren. Maar zodra het lezende proces de nieuwe waarde heeft geconstateerd dan zal het bij ieder volgende inspectie, even-

tueel nog steeds gelijktijdig met diezelfde toekenning plaatsvindend, wederom de nieuwe waarde moeten constateren (tenzij natuurlijk nieuwe toekenningen inmiddels een andere waarde rechtvaardigen).
Met andere woorden: een waardeverandering die het resultaat is van één enkele schrijfopdracht door het schrijvende proces mag nooit in omgekeerde richting door het lezende proces worden waargenomen.
Vanuit het gezichtspunt van het inspecterende proces vinden waardeveranderingen dus steeds tussen twee inspecties plaats. En omdat de veranderingen niet in omgekeerde richting kunnen worden waargenomen (zelfs niet langs indirekte weg: de communicatie vindt alleen plaats tussen de arbiter en ieder van de processen afzonderlijk) wordt hier hetzelfde effekt bereikt als wanneer de atomiciteit van inspecties en toekenningen zou gelden.
De oplossing is analoog aan de vorige. We merken nog op dat process(j) in deze oplossing alleen dan kan worden toegestaan om de competitie rond de kritieke sectie (opnieuw) aan te gaan wanneer de arbiter de waarde van answered(j) weer op false heeft gezet sinds process(j) de vorige maal zijn kritieke sectie heeft verlaten. De beide volgende diagrammen verbeelden de toekenningen (———) en inspecties (-----) voor het paar (attention(j), answered(j)) respektievelijk gezien door de ogen van process(j) en van de arbiter.



process (j)



arbiter

fig. 1

In de zojuist behandelde oplossing veronderstelden wij het postulaat van de onomkeerbaarheid: een waardeverandering die het resultaat is van één enkele schrijfopdracht door het schrijvende proces kan nooit in omgekeerde richting door het lezende proces worden waargenomen.
We zullen nu die oplossing omvormen zodat zelfs dit postulaat van de onomkeerbaarheid niet meer behoeft te worden aangenomen.

Daartoe introduceren we het array

```
changing : array 0 .. N of boolean
```

met initiële waarde false voor ieder van de array elementen.
We omgeven vervolgens iedere toekenning aan attention(j) met de opdrachten changing(j) := true en changing(j) := false en we omgeven iedere toekenning aan answered(j) met de opdrachten changing(0) := true en changing(0) := false. Ook plaatsen we extra lussen while changing(j) do od respectievelijk while changing(0) do od tussen ieder tweetal waargenomen wijzigingen van attention(j) respectievelijk answered(j).
Blijkbaar kan een verandering van de waarde van attention(j) (of van answered(j)) nu alleen nog maar worden waargenomen als changing(j) (resp. changing(0)) na de vorige waargenomen wijziging van attention(j) (resp. answered(j)) in de toestand false bevonden is. Maar dan is de vorige waargenomen verandering van waarde zeker al tot een goed einde gekomen, omdat tijdens de verandering changing(j) (resp. changing (0)) de waarde true heeft.
De volledig omgevormde oplossing luidt:

```
arbiter : begin i: integer; i := 1;
              repeat
                   if attention(i) then
                      changing(0) := true;
                      answered(i) := true;
                      changing(0) := false;
                      while changing(i) do od;
                      while attention(i) do od;
                      changing(0) := true;
                      answered(i) := false;
                      changing(0) := false;
                      while changing(i) do od
                   fi;
                   i := i + 1; if i > N then i := 1 fi
              forever
         end
```

```
process(j) : while ¬ finished(j) do
                 while answered(j) do od;
                 changing(j) := true;
                 attention(j) := true;
                 changing(j) := false;
                 while changing(0) do od;
                 while ¬ answered(j) do od;
                 critical section;
                 while changing(0) do od;
                 changing(j) := true;
                 attention(j) := false;
                 changing(j) := false;
                 noncritical section
             od
```

Het idee dat de veranderingen van attention(j) en ansewred(j) nooit in omgekeerde richting mogen worden waargenomen is hier zuiver door middel van programmering met behulp van het array changing verwezenlijkt. We behoeven het postulaat van de onomkeerbaarheid hier dus niet meer te veronderstellen. Deze versie heeft bovendien de opmerkelijke eigenschap dat bij gelijktijdige toekenning en waarneming (d.w.z. in niet disjunkte tijdsintervallen) van dezelfde informatiedrager de waarneming een willekeurige logische waarde mag opleveren.

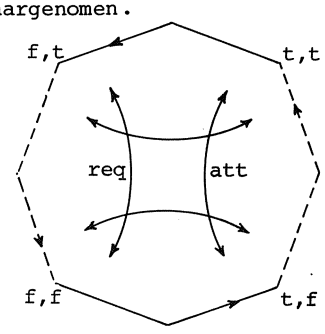
We beschrijven tenslotte nog een kortere oplossing met dezelfde opmerkelijke eigenschap. Ook hier is door middel van programmering voorkomen dat wijzigingen van gemeenschappelijke informatiedragers ooit in omgekeerde richting worden waargenomen. De informatiedrager attention(j) heeft een compagnon request(j) en de informatiedrager answered(j) een compagnon allowed(j). De compagnons vervullen ten opzichte van elkaar de rol van changing(j) in de vorige oplossing. Alle array elementen zijn weer geïnitialiseerd met de waarde false.

```
arbiter : begin i : integer; i := 1;
              repeat
                   if attention(i) then
                      allowed(i) := true;
                      answered(i) := true;
                      while request(i) do od;
                      answered(i) := false;
                      allowed(i) := false
                   fi;
                   i := i + 1; if i > N then i := 1 fi
              forever
          end
```

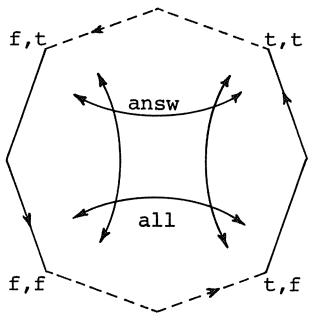
en voor j = 1,...,N

```
process(j) : while ¬ finished(j) do
                  while allowed(j) do od;
                  request(j) := true;
                  attention(j) := true;
                  while ¬ answered(j) do od;
                  critical section;
                  attention(j) := false;
                  request(j) := false;
                  noncritical section
             od
```

De volgende diagrammen verbeelden weer de toekenningen en waarnemingen zoals die door de bril van process(j) respectievelijk van de arbiter worden waargenomen.



process (j)



arbiter

Bij wijze van toelichting bekijken we de wijziging van de waarde van request(j) van true naar false zoals die in de opdracht while request(j) do od door de arbiter zal worden waargenomen. Deze wijziging wordt veroorzaakt door de toekenning request(j) := false in process(j) welke toekenning is omgeven door attention(j) := false en attention(j) := true. Merk op dat de wijzigingen van request(j) alleen gedurende het tijdsinterval dat attention(j) = false plaats vinden. Tenslotte wordt tussen ieder tweetal waargenomen wijzigingen van request(j) de waarde van attention(j) door de arbiter in de toestand true bevonden, zodat bijgevolg geen enkel tweetal waargenomen wijzigingen van request(j) door toekenning(en) in dezelfde cykel kan zijn veroorzaakt.

3. ANDERE SYNCHRONISATIECONCEPTEN

In het artikel Cooperating sequential processes [4] introduceerde DIJKSTRA het nu welbekende synchronisatieconcept van de semafoor.
De semafoor kan worden gebruikt om de kritieke sectie te implementeren, maar kan ook voor andere doeleinden worden toegepast.
Dijkstra definieerde de semafoor onmiddelijk nadat hij in het voorafgaande deel van zijn artikel een oplossing voor het probleem van de implementeerbaarheid van de kritieke sectie uitgaande van de atomiciteit van inspecties en toekenningen had gedemonstreerd. Desalniettemin werden de semafoor en de bijbehorende operaties op de semafoor uitsluitend op informeel beschrijvende wijze ingevoerd en werd geen duidelijke poging ondernomen om een implementatie te vinden analoog aan die van de kritieke sectie. Na Dijkstra's semafoor zijn nog verscheidene andere synchronisatieconcepten voorgesteld. We noemen de belangrijkste: de konditionele kritieke sectie en de monitor. Dergelijke synchronisatieconcepten bleken equivalent te zijn aan de semafoor in die zin dat zij konden worden gebruikt om de semafoor te implementeren en omgekeerd, dat de semafoor voldoende was om het andere concept te verwezenlijken.
Maar merkwaardig genoeg heeft, voor zover mij bekend, niemand nog ooit een implementatie van een van die synchronisatieconcepten getoond welke niet de aanwezigheid van een van de andere concepten vooronderstelde.
Alvorens een oplossing voor de implementatie van de semafoor met behulp van een arbiter te presenteren, beschrijven wij de semafoor en zijn operaties eerst op een meer informele wijze.
Een semafoor is een speciaal type informatiedrager welke uitsluitend kan wor-

den geïnitialiseerd met een niet-negatieve geheeltallige waarde en waarop na initialisatie slechts twee soorten operaties toepasbaar zijn, nl. de wait operatie en de signal operatie. De beide operaties wait en signal zijn alleen toepasbaar op informatiedragers van het type semafoor en worden geacht atomaire opdrachten te zijn zodat wederzijdse uitsluiting van semafooroperaties moet worden aangenomen. Gegeven die wederzijdse uitsluiting van de uitvoering van semafooroperaties kan hun effect dan als volgt beschreven worden

```
wait(s) : s := s - 1;
          if s < 0 then go to sleep on s fi
```

en

```
signal(s) : s := s + 1;
            if s ≤ 0 then wake up one of the sleepers on s fi
```

Deze beschrijving verschilt enigszins van de oorspronkelijke beschrijvingen van Dijkstra. Het belangrijkste verschil is dat de waarde van de semafoor negatief kan worden. Aldus hebben we een eenvoudige mogelijkheid om de aanwezigheid van wachtende processen te constateren. Omdat de waarde van de semafoor toch niet toegankelijk is voor de individuele processen is dit verschil voor hen van geen enkele betekenis.

De nu volgende implementatie van de semafoor maakt gebruik van de arrays

```
waiting     : array 1 .. N of boolean;
signalling  : array 1 .. N of boolean;
answered    : array 1 .. N of boolean;
interrupted : array 1 .. N of boolean
```

Alle array elementen zijn weer geïnitialiseerd op de waarde false.

De arbiter gedraagt zich als volgt:

```
arbiter : begin i,k: integer; i := 1;
              repeat
                  if waiting(i) and ¬ interrupted(i) then
                     s := s - 1;
                     if s < 0 then
                        interrupted(i) := true
                     else
                        answered(i) := true;
                        while waiting(i) do od;
                        answered(i) := false
                     fi
                  else
                     if signalling(i) then
                        s := s + 1;
                        if s ≤ 0 then
                           choose k such that interrupted(k);
                           interrupted(k) := false;
                           answered(k) := true;
                           while waiting(k) do od;
                           answered(k) := false
                        fi;
                        answered(i) := true;
                        while signalling(i) do od;
                        answered(i) := false
                     fi
                  fi;
                  i := i + 1; if i > N then i := 1 fi
              forever
          end
```

Voor process(j) bestaat wait(s) uit

```
while answered(j) do od;
waiting(j) := true;
while ¬ answered(j) do od;
waiting(j) := false
```

en evenzo bestaat signal(s) uit

```
while answered(j) do od;
signalling(j) := true;
while ¬ answered(j) do od;
signalling(j) := false
```

We hebben deze oplossing weer neergeschreven onder de aanname van het postulaat van de onmkeerbaarheid van wijzigingen van de waarde van gemeenschappelijke informatiedragers door inspectie. Met behulp van de eerder besproken technieken is deze oplossing natuurlijk eveneens zodanig om te vormen dat die aanname achterwege kan blijven.
De opdracht choose k such that interrupted(k) moet worden opgevat als een niet deterministische toekenning aan de informatiedrager k.

De oplossing is gepresenteerd alsof er op een aktieve, waakzame manier wordt gewacht in de lussen while ¬ answered(j) do od. Maar met een kleine verandering van de betekenis van sommige opdrachten kan de oplossing ook dienen als een beschrijving van de implementatie van de semafoor met een passieve manier van wachten in de wait operatie. Daartoe veronderstellen we een interruptiemechanisme zodanig dat interrupted(j) wordt geassocieerd met een interruptie-conditie voor process(j), dat answered(j) wordt geassocieerd met een niet-interrumpeerbare conditie voor process(j) en alle overige toestanden met een interrumpeerbare conditie.

Conclusies:
Het gebruik van een aktieve systeemcomponent voor de realisatie van synchronisatieconcepten heeft een aantal duidelijke voordelen
- het maakt een direkte onderlinge communicatie tussen alle rivaliserende processen onderling onnodig door de communicatie te beperken tot de arbiter en ieder van de concurrerende processen afzonderlijk;
- de arbiter kan taken verrichten welke heel moeilijk of misschien zelfs onmogelijk te verrichten zijn door de individuele processen afzonderlijk, zoals bijvoorbeeld het dwingend beschikbaar maken van een processor voor de uitvoering van een proces in niet-interrumpeerbare conditie of het interrumperen van een proces dat bezig is aan een atomaire handeling (kritieke sectie) en het daarbij weer vrij maken van de toegang tot die atomaire handeling (kritieke sectie) zoals dat bij de implementatie van de semafoor nodig is.

De arbiter kan worden gezien als een systeem structurerend concept ongeveer op dezelfde wijze als de monitor een systeem structurerend concept is zodra de daartoe benodigde synchronisatieconcepten eenmaal zijn geïmplementeerd.

LITERATUUR

[1] DIJKSTRA, E.W., *Solution of a problem in concurrent programming control*, CACM 8 (1965) 569.

[2] KNUTH, D.E., *Additional comments on a problem in concurrent programming control*, CACM 9 (1966) 321-322.

[3] EISENBERG, M.A. & M.R. McGUIRE, *Further comments on Dijkstra's concurrent programming control problem*, CACM 15 (1972) 999.

[4] DIJKSTRA, E.W., *Co-operating sequential processes*, in Programming languages (Genuys, ed), Academic Press, (1968).

[5] BRINCH HANSEN, P., *Operating system principles*, Prentice-Hall, Englewood Cliffs, N.J. (1973).

[6] DIJKSTRA, E.W., *Hierarchical ordering of sequential processes*, Acta Inf. 1 (1972) 115-138.

[7] HOARE, C.A.R., *Towards a theory of parallel programming*, in Operating systems techniques, Academic Press (1972) 11-25.

[8] HOARE, C.A.R., *Monitors: an operating system structuring concept*, CACM 17 (1974) 549-557.

[9] LAMPORT, L., *A new solution of Dijkstra's concurrent programming problem*, CACM 17 (1974) 453-455.

[10] GOEMAN, H.J.M. *The arbiter: a system structuring concept for implementing synchronizing primitives*, Rapport Instituut voor Toegepaste Wiskunde en Informatica, HGO/75/08 (aug. 1975).

UITGAVEN IN DE SERIE MC SYLLABUS

Onderstaande uitgaven zijn verkrijgbaar bij het Mathematisch Centrum, 2e Boerhaavestraat 49 te Amsterdam-1005, tel. 020-947272.

MCS 1.1 F. GÖBEL & J. VAN DE LUNE, *Leergang Besliskunde, deel 1: Wiskundige basiskennis*, 1965. ISBN 90 6196 014 2.

MCS 1.2 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 2: Kansberekening*, 1965. ISBN 90 6196 015 0.

MCS 1.3 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 3: Statistiek*, 1966. ISBN 90 6196 016 9.

MCS 1.4 G. DE LEVE & W. MOLENAAR, *Leergang Besliskunde, deel 4: Markovketens, en wachttijden*, 1966. ISBN 90 6196 017 7.

MCS 1.5 J. KRIENS & G. DE LEVE, *Leergang Besliskunde, deel 5: Inleiding tot de mathematische besliskunde*, 1966. ISBN 90 6196 018 5.

MCS 1.6a B. DORHOUT & J. KRIENS, *Leergang Besliskunde, deel 6a: Wiskundige programmering 1*, 1968. ISBN 90 6196 032 0.

MCS 1.7a G. DE LEVE, *Leergang Besliskunde, deel 7a: Dynamische programmering 1*, 1968. ISBN 90 6196 033 9.

MCS 1.7b G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7b: Dynamische programmering 2*, 1970. ISBN 90 6196 055 X.

MCS 1.7c G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7c: Dynamische programmering 3*, 1971. ISBN 90 6196 066 5.

MCS 1.8 J. KRIENS, F. GÖBEL & W. MOLENAAR, *Leergang Besliskunde, deel 8: Minimaxmethode, netwerkplanning, simulatie*, 1968. ISBN 90 6196 034 7.

MCS 2.1 G.J.R. FÖRCH, P.J. VAN DER HOUWEN & R.P. VAN DE RIET, *Colloquium stabiliteit van differentieschema's, deel 1*, 1967. ISBN 90 6196 023 1.

MCS 2.2 L. DEKKER, T.J. DEKKER, P.J. VAN DER HOUWEN & M.N. SPIJKER, *Colloquium stabiliteit van differentieschema's, deel 2*, 1968. ISBN 90 6196 035 5.

MCS 3.1 H.A. LAUWERIER, *Randwaardeproblemen, deel 1*, 1967. ISBN 90 6196 024 X.

MCS 3.2 H.A. LAUWERIER, *Randwaardeproblemen, deel 2*, 1968. ISBN 90 6196 036 3.

MCS 3.3 H.A. LAUWERIER, *Randwaardeproblemen, deel 3*, 1968. ISBN 90 6196 043 6.

MCS 4 H.A. LAUWERIER, *Representaties van groepen*, 1968. ISBN 90 6196 037 1.

MCS 5 J.H. VAN LINT, J.J. SEIDEL & P.C. BAAYEN, *Colloquium discrete wiskunde*, 1968. ISBN 90 6196 044 4.

MCS 6 K.K. KOKSMA, *Cursus ALGOL 60*, 1969. ISBN 90 6196 045 2.

MCS 7.1 *Colloquium Moderne rekenmachines, deel 1*, 1969. ISBN 90 6196 046 0.

MCS 7.2 *Colloquium Moderne rekenmachines, deel 2*, 1969. ISBN 90 6196 047 9.

MCS 8 H. BAVINCK & J. GRASMAN, *Relaxatietrillingen*, 1969. ISBN 90 6196 056 8.

MCS 9.1 T.M.T. COOLEN, G.J.R. FÖRCH, E.M. DE JAGER & H.G.J. PIJLS, *Elliptische differentiaalvergelijkingen, deel 1*, 1970. ISBN 90 6196 048 7.

MCS 9.2 W.P. VAN DEN BRINK, T.M.T. COOLEN, B. DIJKHUIS, P.P.N. DE GROEN, P.J. VAN DER HOUWEN, E.M. DE JAGER, N.M. TEMME & R.J. DE VOGELAERE, *Colloquium Elliptische differentiaalvergelijkingen, deel 2*, 1970. ISBN 90 6196 049 5.

MCS 10 J. FABIUS & W.R. VAN ZWET, *Grondbegrippen van de waarschijnlijkheidsrekening*, 1970. ISBN 90 6196 057 6.

MCS 11 H. BART, M.A. KAASHOEK, H.G.J. PIJLS, W.J. DE SCHIPPER & J. DE VRIES, *Colloquium Halfalgebra's en positieve operatoren*, 1971. ISBN 90 6196 067 3.

MCS 12 T.J. DEKKER, *Numerieke algebra*, 1971. ISBN 90 6196 068 1.

MCS 13 F.E.J. KRUSEMAN ARETZ, *Programmeren voor rekenautomaten; De MC ALGOL 60 vertaler voor de EL X8*, 1971. ISBN 90 6196 069 X.

MCS 14 H. BAVINCK, W. GAUTSCHI & G.M. WILLEMS, *Colloquium Approximatietheorie*, 1971. ISBN 90 6196 070 3.

MCS 15.1 T.J. DEKKER, P.W. HEMKER & P.J. VAN DER HOUWEN, *Colloquium Stijve differentiaalvergelijkingen, deel 1*, 1972. ISBN 90 6196 078 9.

MCS 15.2 P.A. BEENTJES, K. DEKKER, H.C. HEMKER, S.P.N. VAN KAMPEN & G.M. WILLEMS, *Colloquium Stijve differentiaalvergelijkingen, deel 2*, 1973. ISBN 90 6196 079 7.

MCS 15.3 P.A. BEENTJES, K. DEKKER, P.W. HEMKER & M. VAN VELDHUIZEN, *Colloquium Stijve differentiaalvergelijkingen, deel 3*, 1975. ISBN 90 6196 118 1.

MCS 16.1 L. GEURTS, *Cursus Programmeren, deel 1: De elementen van het programmeren*, 1973. ISBN 90 6196 080 0.

MCS 16.2 L. GEURTS, *Cursus Programmeren, deel 2: De programmeertaal ALGOL 60*, 1973. ISBN 90 6196 087 8.

MCS 17.1 P.S. STOBBE, *Lineaire algebra, deel 1*, 1974. ISBN 90 6196 090 8.

MCS 17.2 P.S. STOBBE, *Lineaire algebra, deel 2*, 1974. ISBN 90 6196 091 6.

MCS 17.3 N.M. TEMME, *Lineaire algebra, deel 3*, 1976. ISBN 90 6196 123 8.

MCS 18 F. VAN DER BLIJ, H. FREUDENTHAL, J.J. DE IONGH, J.J. SEIDEL & A. VAN WIJNGAARDEN, *Een kwart eeuw wiskunde 1946-1971, Syllabus van de Vakantiecursus 1971*, 1974. ISBN 90 6196 092 4.

MCS 19 A. HORDIJK, R. POTHARST & J.TH. RUNNENBURG, *Optimaal stoppen van Markovketens*, 1974. ISBN 90 6196 093 2.

MCS 20 T.M.T. COOLEN, P.W. HEMKER, P.J. VAN DER HOUWEN & E. SLAGT, *ALGOL 60 procedures voor begin- en randwaardeproblemen*, 1976. ISBN 90 6196 094 0.

	MCS 21	J.W. DE BAKKER (red.), *Colloquium Programmacorrectheid*, 1975. ISBN 90 6196 103 3.
*	MCS 22	R. HELMERS, F.H. RUYMGAART, M.C.A. VAN ZUYLEN & J. OOSTERHOOF, *Asymptotische methoden in de statistiek*, 1976. ISBN 90 6196 104 1.
	MCS 23.1	J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 1*, 1976. ISBN 90 6196 105 X.
*	MCS 23.2	J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 2*, 1976. ISBN 90 6196 115 7.
*	MCS 24.1	P.J. VAN DER HOUWEN, *Numerieke integratie van differentiaalvergelijkingen, deel 1: Eenstapsmethoden* , 1974. ISBN 90 6196 106 8.
	MCS 25	*Colloquium Structuur van Programmeertalen*, 1976. ISBN 90 6196 116 5.
	MCS 26.1	N.M. TEMME (red.), *Nonlinear Analysis, volume 1*, 1976. ISBN 90 6196 117 3.
	MCS 26.2	N.M. TEMME (red.), *Nonlinear Analysis, volume 2*, 1976. ISBN 90 6196 121 1.
	MCS 27	M. BAKKER, P.W. HEMKER, P.J. VAN DER HOUWEN, S.J. POLAK & M. VAN VELDHUIZEN, *Colloquium Discreteringmethoden*, 1976. ISBN 90 6196 124 6.
*	MCS 28	N.M. TEMME (red.), *Nonlinear Diffusion Problems*, 1976. ISBN 90 6196 126 2.
*	MCS 29.1	J.C.P. BUS (red.), *Numerieke Programmatuur, deel 1*, 1976. ISBN 90 6196 128 9.
	MCS 31	J.H. VAN LINT (red.), *Inleiding in de Coderingstheorie*, 1976. ISBN 90 6196 136 X.
	MCS 32	L. GEURTS (red.), *Colloquium Bedrijfssystemen*, 1976. ISBN 90 6196 137 8.

De met een * gemerkte uitgaven moeten nog verschijnen.