A MULTIPROCESSOR SYSTEM WITH MULTITASKING FACILITIES

I. JURCA

A MULTIPROCESSOR SYSTEM WITH MULTITASKING FACILITIES

ISBN 90 6231 038 9

A MULTIPROCESSOR SYSTEM WITH MULTITASKING FACILITIES

PROEFSCHRIFT

ter verkrijging van de graad van Doctor in de Technische Wetenschappen aan de Technische Hogeschool Delft, op gezag van de Rector Magnificus Prof. Ir. L. Huisman, voor een commissie aangewezen door het college van dekanen te verdedigen op woensdag 8 juni 1977 te 14.00 uur

door

IOAN JURCA

elektrotechnisch ingenieur
geboren te Tapia – Roemenië

1977
Dutch Efficiency Bureau – Pijnacker

Dit proefschrift is goedgekeurd door de promotor
Lector Ir. G. L. Reijns

Pentru
Maria şi Adriana,
cu dragoste.

ACKNOWLEDGEMENTS

This thesis resulted from the work done with the support of a scholarship granted by the Romanian Ministry of Education, for which I wish to express my deep gratitude. The support of the Dutch Ministry of Education and Sciences in meeting the printing costs is gratefully acknowledged.

I am profoundly indebted to ir.F.P.C.Pauwels, of the Laboratory for Information Processing Machines of the Delft University of Technology, for his continuous guidance and encouragement.

Special thanks are due to Mrs.J.B.Cavender-Bruner for her attempts to correct the "English" in which the thesis was initially written.

Finally, I wish to acknowledge the support received from the Electrical Engineering Department of the Delft University of Technology during my stay in the Netherlands. I am grateful to all those who took from their time to help me bring this work to an end.

I.Jurca

LIST OF ABBREVIATIONS

Abbreviations used in the text.

AST	-	Active Segment Table
ASTE	-	Active Segment Table Entry
ATR	-	Address Translation Register
BR	-	Base Register
CCS	-	Central Capability Segment
CPU	-	Central Processing Unit
CPUQ	-	CPU Queue
CR	-	Card Reader
CR i	-	Capability Register i
DBR	-	Data Base Register
ER	-	Execution Ring
FIFO	-	First In First Out
GCR	-	Generator for Card Reader
GT	-	Generator for Terminal
I/O, IO	-	Input/Output
KST	-	Known Segment Table
LP	-	Line Printer
LPAV	-	Line Printer Available
MB	-	Memory Block
MM	-	Memory Module
MT	-	Magnetic Tape
PC	-	Program Counter
PCB	-	Process Control Block
PCCT	-	Process Central Capability Table
PCS	-	Primary Capability Segment
PE	-	Processing Element
PEM	-	Processing Element Memory
PPM	-	Parallel Programs Manager
PSB	-	Processor Status Block
PST	-	Process Segment Table
PSW	-	Program Status Word
PT	-	Page Table

SCT	-	System Capability Table
SCTE	-	System Capability Table Entry
SDW	-	Segment Descriptor Word
SPT	-	System Page Table
TST	-	Test and Set

Abbreviations used in references

AFIPS	-	American Federation of Information Processing
CACM	-	Communications of the Association for Computing Machinery
FJCC	-	Fall Joint Computer Conference
IEEE	-	Institute of Electrical and Electronic Engineers
IEEE Tr. on C.	-	IEEE Transactions on Computers
IEEE Tr. on S.E.	-	IEEE Transactions on Software Engineering
NCC	-	National Computer Conference
R.A.I.R.O.	-	Revue Française d'Automatique, Informatique et Recherche Opérationelle
SJCC	-	Spring Joint Computer Conference

CONTENTS

INTRODUCTION

The starting point of the work reported in this thesis was a study of the possibilities of increasing the performance of a computer system by structural means. This is an alternative to increasing the performance by using faster basic circuitry in a conventional structure. During the study, the scope of the potential application areas of the various examined structures was also taken into account.

All structures examined in this initial study show a certain type of parallelism in their operation. The concept of "parallelism" has been interpreted here in a very wide sense. In this sense, pipeline computers, array and associative computers, and multiprocessor systems can all be considered to include parallelism. The study has led to the conclusion that multiprocessor systems offer the greatest flexibility and, therefore, the largest potential area of applications. For these computers, an essential problem is the availability of a suitable operating system.

The outcome of this preparatory work pointed to a further investigation of the multiprocessor systems and in particular of their operating systems. In this context, the means for expressing the coordination between processes and, more generally, for a systematic design of the operating system are of outstanding importance.

The programming language Concurrent Pascal is regarded by the author as one of the best available tools for the design of an operating system and has, therefore, been selected for the example system developed in this thesis. Concurrent Pascal allows a flexible hierarchical organization of

the system and supplies powerful operations for process coordination. During the design, some features that are specific to multiprocessor systems have received special attention. These features stem from the possibility that two or more processors can try to access the same data item simultaneously.

On a somewhat more generalized level, the thesis contains the functional design of a complete operating system for a multiprocessor. The system is supposed to be used in a combined batch and time-sharing processing environment. A particular feature of this system is the possibility of concurrently executing several tasks originating from the same program. This feature is called "multitasking" and a program divided into tasks, a "parallel program."

A simplified simulation model of such a system has been developed and a simulation program has been written using the programming language SIMULA 67. The simulation results indeed show a substantial reduction in the turnaround time of the parallel programs compared to the turnaround time of the same programs executed in a system without multitasking.

The thesis is divided into six chapters.

A review of the initial study on parallelism in computers is presented in Chapter I.

In Chapter 2 the general problems that should be solved in the design of an operating system are introduced and exemplified for various systems.

The design of the operating system kernel is the subject of Chapter 3, and the functional design of the operating system as a Concurrent Pascal program is presented in Chapter 4.

The simulation model and the simulation experiments and results are discussed in Chapter 5.

The last chapter contains a few conclusions and indicates possible directions for further work.

Chapter 1

PARALLELISM IN COMPUTERS

According to the definitions given by Blaauw [6], the architecture of a system is "the functional appearance of the system to the user" and the implementation refers to the system's "inner structure, considered from a logical point of view."

In the attempt to build computers with greater computing power, within the constraints of a certain technology, the implementation has an important role. The basic idea behind all the implementations studied in this chapter is to provide means for the simultaneous (i.e., parallel) execution of several actions.

For specific application areas, e.g., for those involving intensive matrix manipulation, special architectures have been proposed, which make the parallel operations visible to the user. These systems are properly called parallel computers.

In this chapter a few systems are discussed in order to emphasize the technological problems of their implementation and the implications for programming and application areas. At the end of the chapter, the reasons are given for the selection of the multiprocessor systems for a more detailed study. This chapter is based on an earlier report [9] prepared by the author.

1.1. Pipeline computers.

Pipelining is an implementation technique in which an increase in the number of instructions performed by a computer in a time unit is obtained by overlapping the processing of several instructions. In the exemple presented below, the computer has a basically von Neumann-architecture, but the technique of pipelining can also be applied for other architectures.

Among the systems that use this technique are IBM 360/91 [1], MU 5 [8], and Texas Instruments' ASC [14].

Four main phases can be identified in the processing of an instruction: fetch, decode, operand access (when required) and execution. In principle, it is possible to overlap the phases of several successive instructions, i.e., to start the fetch of the next instruction at some time before the processing of the current instruction is completed. This principle is illustrated in Fig. 1.1. The computer completes more instructions in a certain time than a computer without pipelining, although the execution time of the individual instructions is the same.

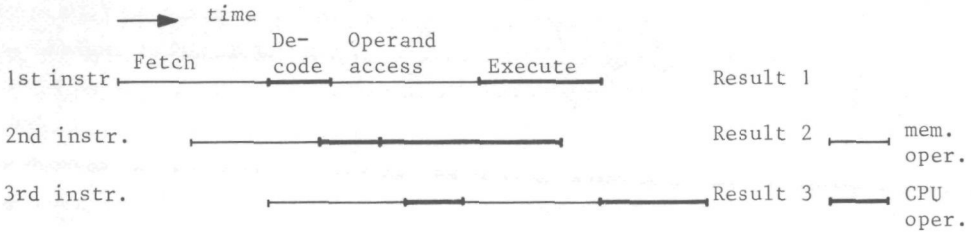


Fig. 1.1. The principle of pipeline operation.

In the text of this chapter, the term control unit (CU) refers to that functional unit of a computer which obtains the successive instructions of a program from the main memory, decodes them and sends the necessary control signals for the instruction execution to the execution units (EU). An execution unit is a functional unit which can perform certain arithmetic and/or logic operations on the data supplied to it, as directed by a CU. The term "processor" or "central processing unit" (CPU) applies to a combination of a CU and one or more execution units directed by the CU.

Taking into account these definitions, it is clear that the main influence of the pipelining approach appears in the structure of a CU, which has to organize the instruction overlapping. However, an efficient implementation requires the other (possibly increased number of) functional units to have the adequate features as well.

A few characteristics of the pipeline computers will now be discussed with the aid of Fig. 1.2, which represents in a simplified form the implementation used for IBM 360/91.

It may be assumed that the instructions of a program are stored in consecutive memory locations. If the memory is not divided into blocks, the fetch of the next instruction cannot start before the memory cycle that fetches the current instruction is completed. An interleaved memory i.e., a memory unit which is divided into separately addressable blocks, and has the consecutive locations in different blocks, removes this limitation.

In the CU a number of buffers must be provided for the storage of the instructions already fetched from memory, but not yet decoded. The unit that controls the access to the memory system (called in IBM 360/91 system the Main Storage Control Element - MSCE -) is also required to provide some address and data buffers. The Storage Conflict Buffers are used to queue addresses to busy storage modules. The presence of buffers is, in fact, an essential characteristic of the pipeline computers. The buffers contribute to the continuity of the information flow and thus to the efficient use of the computer functional units.

The principle of pipelining can be extended to the internal organization of the execution units. An adder, for instance, can be designed in such a way that two additions are simultaneously in progress. Another way to improve the performance of the arithmetic unit is to divide it into smaller functional units which can work independently.

If successive instructions refer to different units, they can be executed simultaneously. The TI ASC computer combines the two approaches: the arithmetic unit is divided into 8 functional units and the execution of an instruction means the successive activation of some of these units.

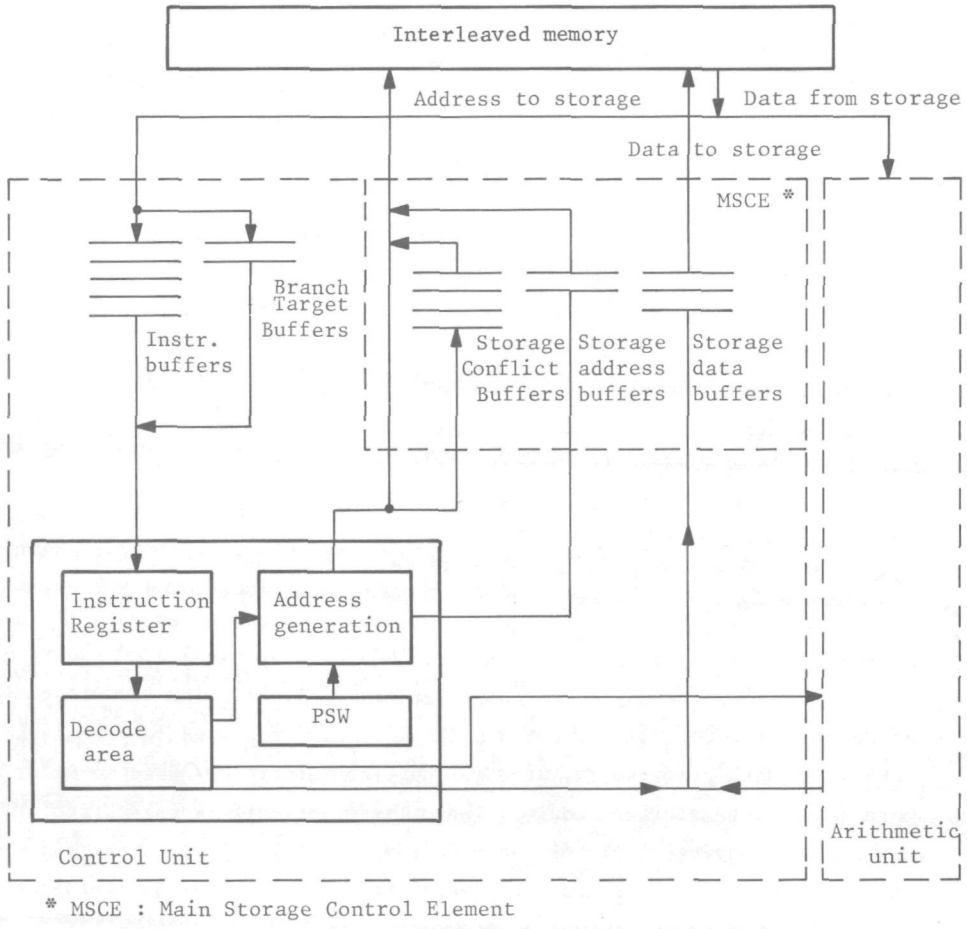


Fig. 1.2. An example of pipeline implementation: IBM 360/91.

The efficiency of the pipelining is strongly influenced by the interdependence of the successive instructions in a program and by the number of branch instructions. When an instruction uses the result of a previous instruction as an operand, it cannot be executed until that result is available which, of course, reduces the overall execution speed. For this reason, techniques like "look-ahead" or "look-aside" [11] have been proposed for determining the independent instructions in a sequence of a certain length. The instruction buffer of the CU stores the instructions under examination.

A further complication is introduced by the branch instructions, because it is not possible to know in advance which path will be chosen by such instructions. Therefore, the IBM 360/91 is equipped with a set of "branch target buffers" (see Fig. 1.2) in addition to the instruction buffers. When a branch instruction is recognized, the CU requests from the main memory two instructions starting at the branch target in addition to the instructions following the branch in the normal sequence. Obviously, the execution of the instructions following the branch in either of the two paths cannot be started until the branch decision is taken. The delay that would be caused by the access to the memory is reduced, because even when the target path is selected, the required instructions are already present in the CU.

Another source of delays in pipeline computers is the interrupts. The contents of the instruction buffers should be completely changed after an interrupt, which causes a discontinuity in the operation of the processor. However, if the interrupts are relatively infrequent their influence on the performance is not significant.

In conclusion, it must be noted that pipelining is an implementation technique, not a particular computer architecture. This technique has been used in computers with different architectures, as is shown by IBM 360/91 and MU5. Most of the computers that have applied this technique have a general-purpose architecture and can be classified as medium-to-large or large systems. From a theoretical point of view [13], the main difficulty in obtaining an efficient implementation consists in the identification of the independent instructions in a sequence of a certain length.

Generally, pipelining is not a modular technique, and therefore it offers little flexibility for further improvements once a system is completely built. The technique does not require any special functions to be incorporated in the operating system. It will profit however, from careful programming which can result in a higher proportion of independent instructions. The pipeline technique is not recommended in systems with frequent interrupts, where it cannot significantly improve the performance. There is also a relation between the characteristics of a computer architecture, such as the type of instructions and the frequency of branch instructions, and the efficiency of a pipeline implementation.

1.2. *Parallel computers.*

There are some applications, for instance, matrix manipulation, image processing and information sharing, in which the same operation must be performed simultaneously on large sets of related data. This fact can be made apparent in the architecture of a computer designed with such applications in mind. In the literature, the designation "parallel computers" is specifically applied to these computers.

Two classes of architectures are discussed in this section: array computers and associative computers.

1.2.1. Array computers.

The best-known design and implementation of an array computer is the ILLIAC IV [2]. A block diagram of this computer is presented in Fig. 1.3.

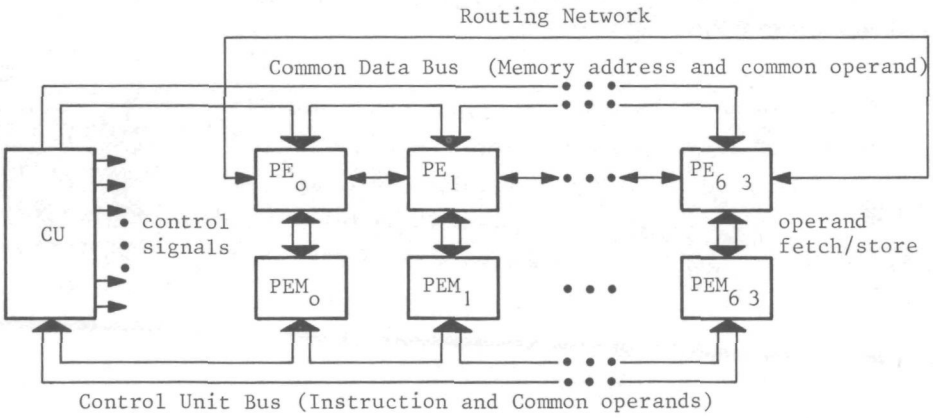


Fig. 1.3. The block diagram of an array computer: ILLIAC IV.

Since the array processors must execute the same operation on multiple data sets, a single CU is used to direct a number of identical execution units. There are 64 execution units, called Processing Element (PE), in ILLIAC IV. When CU decodes an instruction, it sends the control signals to all the processing elements. However,each PE must have the possibility of deciding whether it should execute the decoded instruction or not.

Some of the units must remain idle during certain instructions, because of the results obtained in previous instructions or because the logic of the program requires it.

One of the basic problems in the design and implementation of the array computers is the interconnection of the EUs and the memory. Each instruction operates on a large number of operands, and therefore it is essential that these operands are fetched from the memory in as short a time as possible. A description of the solution adopted by the ILLIAC IV designers follows.

Each PE is provided with a block of memory, denoted as Processing Element Memory (PEM), which contains both data and instructions and which can be directly addressed by that PE. A "routing network" connects the processing elements with each other. Each PE has direct connections with four neighbors. If the 64 PEs are numbered 0,1,2,...,n,...,63, then the element n is connected with n-8, n-1, n+1 and n+8. This pattern of interconnection allows the execution of an exchange of information between any two PEs in at most 7 steps, by steps of 1 or 8 elements. A PE obtains access to the PEM of another processing element, exclusively via the routing network.

The CU has the ability to read the status bits of all PEs and, thereby, to monitor the status of a certain operation. The CU also contains an instruction buffer with a capacity of 128 instructions, which is assumed large enough to store the instructions of the inner loop of many programs. In the programs where this assumption is valid, after the initial loading of the buffer, the instructions for the next cycles of the loop are fetched from the buffer with a minimal delay.

The successive instructions of a program are stored in successive PEMs and due to the physical arrangement of the PEMs, 8 instructions can be accessed simultaneously.

Special programming problems arise for array computers. For ILLIAC IV a high-level language, TRANQUIL, has been developed, which includes some statements for specifying the memory allocation and has a control structure similar to that of ALGOL. Inefficiencies result when the problem data structure does not match exactly the requirements of the PEMs arrangement. An example of this is the processing of a 65x65 matrix in a system with 64 PEs. Algorithms have been studied [10] to reduce this type of inefficiencies.

While useful in particular applications, the array processors cannot be easily adapted to the requirements of a general computing environment. They need a supplementary programming effort and the development of parallel algorithms for the problems intended to be solved on such computers. Nevertheless, the research in this field continues, the structure of the interconnection between processing elements and memory modules receiving much attention in the theoretical studies.

1.2.2. Associative computers

The associative computers are based on the availability of an "associative memory" (also called "content-addressable memory"). A conventional (random-access) memory used the address of a word in order to deliver the word contents. An associative memory uses the (partial) contents that a word should have in order to find the address of such a word. This implies, of course, that the associative memories have a much more complex memory cell than the conventional memories.

In most of the proposed architectures, the associative memory is exclusively used for storing data. A separate memory, built with conventional components is then provided for storing this programs.

The principle of the associative computers is explained with the aid of Fig. 1.4. Specific examples are taken from the STARAN system, described in Ref. [3] .

The memory matrix consists of n words of m bits each. Usually, m has a larger value than in conventional memories (m=256 for STARAN) and, at least in the current proposals, n is not larger than 4 k.

A number of registers are provided which assist the memory operation. The mask register indicates the contents (or, more often, a part of the contents) that the requested memory word must have. When such a word is identified, its contents is transferred into the data register. The word mask register specifies the memory word in which a search operation must be performed. This register allows a partial search of the memory, an operation which is required for complex search conditions that must be performed in steps (e.g., a search on the condition "between limits"). In the same type of operations, the match response register indicates in which words a certain condition is satisfied. The contents of this register is transferred into the word mask register for the next search.

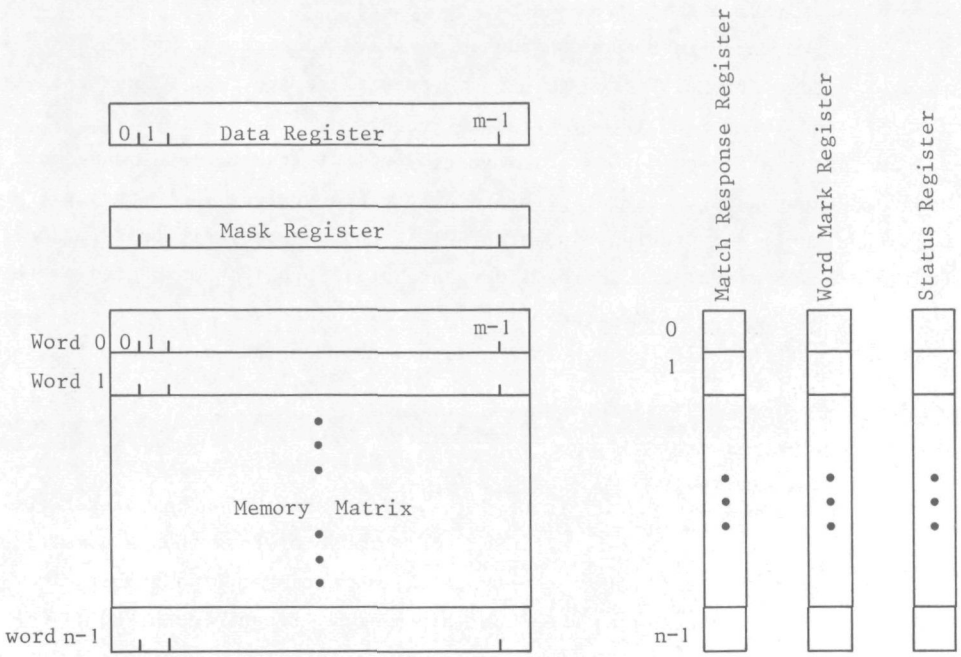


Fig. 1.4. The principle of an associative computer.

The status register shows which memory words are free at any moment. It will be used during the write operations to select a free word.

A search operation, which is an intrinsic property of the associative memories, is the basic operation of the associative computers. Supplementary arithmetic and logic operations should be added. Several approaches to the structure of an arithmetic unit for these computers have been investigated. A "fully parallel system" is obtained if processing capabilities are provided in each memory cell. Alternatively,it is possible to provide a bit serial processing unit, which can perform an operation only on one bit of each memory word at a time.

In the STARAN computer an intermediate solution has been adopted: 8 bits out of a word of 256 bits are processed in parallel. This solution is a compromise between the high cost of a fully parallel unit and the low speed of a serial unit.

The instruction list of an associative computer, like that of an array computer, differs fundamentally from the instruction list of a von Neumann-type computer. But in the associative computers this difference is even

more clearly marked than in the array computers.

A problem which is common to the associative and array computers is the need of a conventional system to execute their software. For STARAN this is a PDP-11 and for ILLIAC IV, a Burroughs B6500.

The associative computers are primarily useful in data base management and image processing. Other possible applications include fast Fourier transformation (FFT) and problems in which FFT is encountered, air traffic control, and radar signal processing. There are still problems connected to the cost of constructing a large associative memory. Moreover, these computers are even less suitable for general applications than the array computers.

1.3. Multiprocessor Systems.

The term "multiprocessor" is used in various senses in the literature, as shown for instance in Ref. [7]. For the purpose of this thesis a multiprocessor system is defined as a computer which consists of a number of identical CPUs connnected to a common main memory. Because each CPU possesses a CU and an arithmetic unit, a multiprocessor system is capable of simultaneously executing several independent programs or programs that are partially independent, but cooperate in solving a certain problem. The CPUs (processors) have a general-purpose architecture.

Modularity is an obvious characteristic of the multiprocessor systems. The computing power of such a system can be increased by adding more CPUs, provided the CPU-memory interconnection scheme (see Fig. 1.5) is appropriately designed. The design of a suitable interconnection scheme is, in fact, one of the most important stages in the development of the hardware of a multiprocessor system. An adequate design must ensure that:

- all CPUs can obtain access to all memory modules;
- the delay caused by memory accesses does not produce a substantial degradation of the system performance;
- adding new processors or memory modules produces only an almost linear increase in the complexity and the cost of the scheme;
- the total cost of the scheme is not prohibitive.

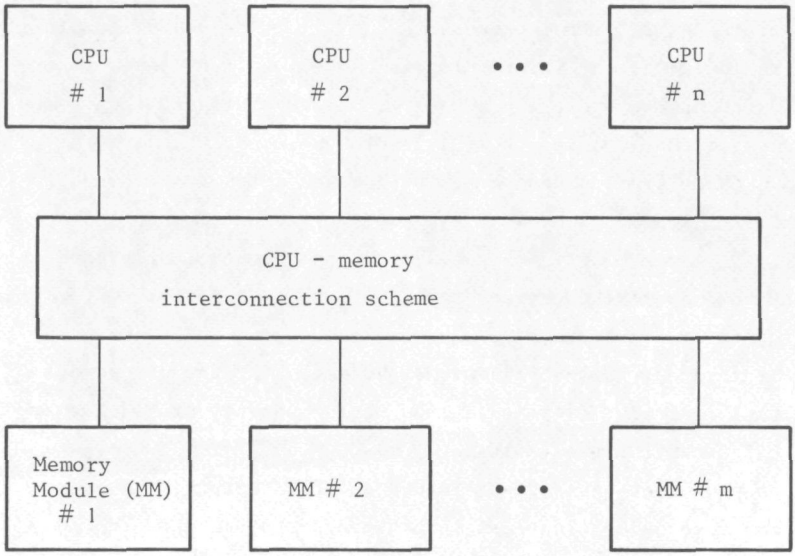


Fig. 1.5. Block diagram of a multiprocessor system.

A study of some implementations for the processors-memory interconnection scheme and a proposal for a modular design of such a scheme can be found in Ref. [6]. The recent availability of inexpensive microprocessors has given new impetus to this field of research [12].

The existence of more than one CPU in the system can be made visible in the architecture of the system, by offering the users the possiblity of dividing their programs into parallel executable tasks and specifying the relationship between the tasks. Alternatively, the software (specifically the compilers) of a multiprocessor system can be written in such a way that parallelism in programs is automatically recognized.

It should be noted that this kind of parallelisms differs from that encountered in pipeline, associative or array computers. A program task usually consists of an entire routine or even of a group of routines. The parallelism is then apparent at a level higher than the instruction level.

Regardless of the architecture of a multiprocessor system (i.e., whether the parallelism at task level is made visible to the programmer or not), the operating system of such a computer must ensure an efficient use of the system resources. The structure of the operating system will not differ essentially from that of a system for a multiprogrammed computer.

However, the existence of several processors adds a new degree of complexity, especially in manipulating the information which defines the system status. A methodical design of the operating system is a more stringent requirement for a multiprocessor than for any other type of system.

The multiprocessor systems have a built-in capability for "fault-tolerance" and "graceful degradation" of the system performance. When a processor fails, the software can re-configure the system and continue to operate at reduced performance. Special "recovery routines" should be provided for this purpose. The identical structure of the system processors is an essential feature in obtaining the "fail-soft" operation. Note that the processors may have different implementations (and thus performances), provided they have an identical architecture.

Because each processor has a general-purpose architecture, a multiprocessor system can also be considered a general-purpose computer. The computing power and/or the memory capacity required by a certain application can be satisfied by a modular expansion of the system configuration. An increase in the number of processors also means in increase in the complexity of the system management and, thus, in the system overhead. Therefore, the useful system computing power is smaller than the sum of the computing powers of the constituent processors.

In conclusion,it can be stated that the multiprocessor systems offer, among the studied configurations, the greatest flexibility in structuring a computer that will correspond to the user's needs, provided an adequate operating system is available. For this reason it has been decided to proceed with a more detailed study of the multiprocessors and, specifically, of the appropriate design of their operating systems.

References

1. Anderson, D.W., et al., "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," IBM Journal of R&D, Vol.11, No.1 (January 1967), pp. 8-24.

2. Barnes, G.H., et al., "The ILLIAC IV Computer," IEEE Tr. on C., Vol.17 No.8 (August 1968), pp. 746-757.

3. Batcher, K.E., "STARAN Parallel Processor Hardware," AFIPS National Computer Conference, 1974, pp. 405-410.

4. Bell, J., et al., "An Investigation of Alternative Cache Organizations," IEEE Tr. on C., Vol.23, No.4 (April 1974), pp. 346-351.

5. Blaauw, G., "Computer Architecture," Elektronische Rechenanlagen, Vol.14, No. 4, 1972, pp. 154-159.

6. Davis, R.L., et al., "A Building Block Approach to Multiprocessing," AFIPS Proceedings, SJCC, 1972, pp. 685-703.

7. Enslow, P.H. (ed.,) Multiprocessing and Parallel Processing, J. Wiley & Sons, New York, 1974.

8. Ibett, R.N., "The MU-5 Instruction Pipeline," The Computer Journal, February 1972, pp. 42-50.

9. Jurca, I., "Parallelism in Computers and its Relationship with Performance and Application Areas of Computer Systems," Internal Report 051560-44 (1975)05, Laboratory for Information Processing Machines, TH Delft.

10. Kuck, D.J., "ILLIAC IV Software and Application Programs," IEEE Tr. on C., Vol. 17, No. 8 (August 1968), pp. 758-770.

11. Lee, F., "Study of 'Look-Aside' Memory," IEEE Tr. on C., Vol. 18, No. 11, (Nov. 1969), pp. 1062-1064.

12. Swan, R.J., et al., "The Structure and Architecture of a Cm*; a Modular, Multi-Microprocessor," Comp. Sci. Dept., Carnegie-Mellon University, August 1976.

13. Tjaden, G.S. and Flynn, M.J., "Detection and Execution of Independent Instructions," IEEE Tr. on C. Vol.19, No.10, (Oct. 1970) pp. 889-895.

14. Watson, J., "The TI ASC - A Highly Modular and Flexible Super Computer Architecture," AFIPS Proceedings, FJCC, 1972, pp. 221-228.

Chapter 2.

BASIC CONCEPTS IN OPERATING SYSTEM DESIGN.

In the evolution of computers, several important concepts common to the majority of existing operating systems, general-purpose or dedicated, have been identified and can serve as a basis for the design of new systems.

In a very general sense, these concepts are related to three main areas:

a) process coordination,

b) resource allocation and scheduling,

c) information addressing and accessing.

Each of these areas will be briefly discussed in this chapter, the most relevant concepts defined, and some examples given to highlight specific aspects of the concepts.

2.1. *Process coordination*

The activity taking place at a certain moment in a computer system can be thought of as consisting of a number of asynchronous processes. A process is a series of strictly sequentially-performed operations that result in the fulfillment of a desired task. In other words, a process can be defined as consisting of:

- a collection of programs,
- a collection of data on which the programs operate,
- the actual, sequential execution of the programs.

The most important feature of a process is the sequential nature of its operations. A process also has the theoretical property that, provided enough computer resources are available and programs appropriately written, its actions can proceed without interference from other processes, simultaneously in execution in the same system (a collection of such processes will subsequently be referred to as "concurrent processes").

However, two important factors lead to some forms of communication be-

tween processes. In the first place, it is uneconomical to provide so many resources that no interference among processes occurs. Generally, a process will not be able to keep all its resources busy most of the time because of the sequential nature of its operations: a new operation cannot start until the previous one has been completed, and one operation usually involves only a small part of the resources.

Connected to the first factor, there is a second: as the amount of resources is limited, the resources must be shared among processes. Thus a cooperation between processes in this sharing is implied.

Therefore, the activity of the computer system must be organized with the necessity of an efficient use of the resources in mind. It can happen that this fact leads to a situation where a "producer/consumer" relation exists between processes: results produced by one process are used for the progress of another process (the two processes are said to cooperate). Another form of process interaction is usually called "mutual exclusion": there are instances when a process requires exclusive use of a resource, or otherwise the results will be unpredictable. A simple example of this form of interaction is that of a process requesting access to a location which can also be accessed by other processes; if the process does not obtain exclusive use of the memory location for the duration of the operation, it may happen that the memory content is inadvertently modified while the operation is still in progress.

The design of the computer should provide means for the implementation of the types of process interactions presented above, i.e., means for process coordination.

The computer system will have status information associated with every resource (e.g., for a magnetic tape or a printer, it should specify whether the device is allocated or free; for a buffer used in the transmission of results from one process to another, whether the buffer is empty or full). All process coordination means must enable a process to access the status information of a resource, examine it and possibly modify it, all in a single, indivisible operation. Such an operation ensures that, when a process needs the status information of a resource, the information is always found in the correct state.

The remainder of this section is dedicated to a discussion of several process coordination means.

2.1.1. Test-and-Set Instruction.

At the lowest level of organization (in view of the process coordination), a computer must be designed with a machine instruction that allows the reading, testing and subsequent writing of a memory location as an indivisible operation.

Such an instruction is present in most of the third-generation computers and appears generally under the name "Test-and-Set" (TST). The TST instruction operates as follows: the content of the addressed memory location is read out, and if it is zero, a one-bit flag is set to "1" and, at the same time, a "1" is written back into that memory location. Only after the write operation is completed, does the memory location become available for other processes. If the memory location does not contain a zero, the flag will not be set to one and the location is simply rewritten.

Following the TST instruction, the process that has executed it will have instructions to test the flag and take a decision about how to continue, depending on the value of the flag.

It is easy to see that the TST instruction provides indeed the means required for process coordination. The "mutual exclusion" is attained by associating a memory location with the resource in question. Before using the resource, the requesting process will issue a TST instruction for that memory location. If the memory location is found to be zero, a "1" will be written in that location and the process can continue with the use of the resource. When the resource is no longer needed, the process resets the memeory location to zero. While the memory location is set to "1", it will be understood that a process is using that resource and other processes issuing TST will not be allowed to continue.

The "producer/consumer" relationship is implemented in a similar way. The producer process will signal that it has produced a new item by resetting a memory location to zero. The same memory location is tested (TST) by the consumer process when the item is needed.

While the TST instruction supplies the means for a correct process coordination, it leaves too many details in charge of the programmer writing the process programs. In particular, the programmer must decide what the process will do when it is not allowed to continue. The simplest solution would be to have the process continuously testing the condition until, presumably by the action of another process, the condition is satisfied. This leads, however, to the so-called "busy waiting": the process keeps a CPU busy with

non-productive work. In order to eliminate the busy waiting, it must be possible that the implied process relinquishes the CPU, so that the CPU can be allocated to another process.

The new process might eventually contribute to the fulfillment of the condition expected by the first process.

A solution that possesses the property stated above has been proposed by Dijkstra [5] . This solution makes use of a special kind of variable, called "semaphore."

2.1.2. Semaphores.

A semaphore is an integer variable on which two indivisible operations, called P ("passeren" = pass by, in Dutch) and V ("vrijgeven" = release, in Dutch) are defined.
There is also a queue associated with a semaphore. If S is a semaphore variable, the P and V operations can be defined as follows:

P(S): *if S > 1 then S:=S-1 and continue process*
else the process is put in the queue.

V(S): *S:=S+1;*
if queue is not empty then resume a process
from the queue else continue process that has
issued V(S).

At this point, a discussion about the process status (as seen from the viewpoint of a processor - CPU or peripheral processor -) is in order.
An active process is one currently making progress by having its programs executed. It often happens that a process might be active, but its programs cannot be executed because all the available processors are currently working for other processes: such a process is said to be in the ready state. Finally, there are situations when a process cannot continue because it depends on a signal from another process, which has not yet arrived; the process is then blocked.
A representation of the states of a process and of the transitions between these states is given in Fig. 2.1.

Referring to the definition of the P and V operations, a process in the queue will be blocked. In the V operations when the queue is not empty, it was said that a process in that queue is resumed. This means that its status is changed from "blocked" to "ready" or "active."

The process issuing the V operation makes, in this case, the transition from the active state to the ready state.

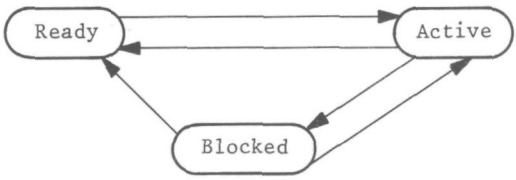


Fig.2.1. Process states and transitions.

The use of the semaphores for process coordination is now exemplified.

Example 1: A producer/consumer relationship. A typical instance of such a relationship is that of an input process(producer) placing new data records in a buffer and a consumer process taking records out of the buffer. Let us consider that the buffer can contain up to n records and that it is initially empty.

Two semaphore variables, NE (not empty) and NF (not full) are associated with the buffer. NE is initialized to zero, and NF is initialized to n. The code for the two processes can now be written as follows:

```
    Producer                    Consumer
repeat                      repeat
    produce item;               P(NE);
    P(NF);                      get item;
    put item;                   V(NF);
    V(NE);                      consume item;
end                         end
```

A producer cannot continue if there is no free space in the buffer. Therefore, after producing a new item, it should test the NF semaphore, which must be strictly positive in order that the producer process can continue; otherwise the producer is blocked.

After placing a new item (record) in the buffer, the producer signals this fact by a V operation on the NE semaphore. If the buffer was previously empty, the consumer process can be in the queue of the NE. And if this is the case, the process can now be resumed.

The consumer starts by executing a P(NE) operation in order to get a new record from the buffer. After taking out the record, the consumer process

executes a V(NF) operation. If the buffer was previously full and the producer process blocked in the queue of the NF, the producer process is now resumed.

Example 2: Mutual exclusion. Consider a set of processes competing for exclusive access to one resource. A semaphore, MX, is associated with that resource and initialized to 1. Before using the resource, a process must issue a P(MX) and if the resource is already being used by another process, MX is zero. Thus the process is placed in the queue of MX; otherwise MX will be reset to zero and the process can continue.

When the process completes its work with the resource, it must execute a V(MX) operation. The resource is made available for other processes.
The process code will thus contain the following section:

```
   .
   .
   .
P(MX);
use the resource;
V(MX);
   .
   .
   .
```

Since the P and V operations are indivisible, it can be proved [8] that the solutions indicated above lead to correct results. With the use of the semaphores, the "busy waiting" problem is reasonably solved. The P and V operations can be implemented in hardware, but most likely they will be implemented as software routines, making use of the TST instruction and/or of the possibility to mask the interrupts in a computer. Although the programmer is relieved of some tedious work, the semaphores remain at a rather primitive level, since the programmer is still concerned with a clear evidence of the semaphores used in his program and of the correct sequence of operations on these semaphores.

Since some of the inconveniences of the semaphores originate in the rather simple nature of the semaphore variable and of the P van V operations, a proposal for the elimination of the semaphore disadvantages should provide more powerful primitive operations and allow them to operate on more complex variables. This is achieved in a solution based on the introduction of shared variables.

2.1.3. Shared variables and critical regions.

A piece of program where a process needs exclusive use of a resource has been called by Dijkstra a critical region. Hoare [12] and Hansen [2] have proposed a new means to express process coordination, based on the concept of shared variable. A variable is said to be shared when two or more processes can modify its values. A shared variable may be of any particular type recognized in a certain programming language. Thus, if v is such a variable, its declaration in a Pascal-like notation can be, for instance,

```
var v : shared integer; or
var v : shared record a,b: real; c:boolean end;
```

By definition, a shared variable is accessible only inside a critical region, and critical regions referring to the same shared variable exclude each other in time.
A critical region using the shared variable v is coded as follows:

```
region v do S ;
```

where S represents the statement(s) of the respective critical region.

The construct, just introduced, allows only a simple representation for the mutual exclusion type of relations between concurrent processes. To provide for process cooperation, a new language primitive, await, has been proposed. It delays a process until the components of a shared variable v satisfy a condition B. The code for such a critical region is:

```
region v do
    begin ... await B ... end;
```

The await primitive must be enclosed by a critical region. If critical regions are nested, the condition B is associated with the innermost enclosing region.

The implementation of these language constructs is illustrated in Fig.2.2.

A process will be put into a "main process queue" Q_v when it needs to enter a critical region associated with a shared variable v that is already being used by another process. In this case the process is in the blocked state. When its turn arrives to access the variable v, the process is taken out of the Q_v, activated and allowed to start its critical region. When the process encounters an await B, the condition B is evaluated. If the condition is true, the process continues. Otherwise, the process is again blocked and put into an "event queue," Q_e.

Geleend op/*Date of issue*:

11 24 24 04 1993

Terugbezorgd op/*Returned on*:

Uiterlijk terugbezorgen op/*Date due*:

22 05 1993

Paraaf/*Initials:*

Documentgegevens/*Received on loan*:

Jurca, I.
MULTIPROCESSOR SYSTEM WITH MULTITASKING FACILITIES

1200425 CBmg bkuitln 248254

Gebruikersgegevens/*Borrower*:

55011 Vries, J.C.W. de---------

Europasingel 106
2661 BW Bergschenhoek

1117.624.5 no

Bibliotheek TU Delft

Magazijnslip/*Picking slip*

Ik wil dit document graag verlengen/*I would like to renew this loan.*

in te vullen door de Bibliotheek TU Delft/*to be completed bij the Delft University of Technology Library*

☐ verlengd tot/*renewed until :*

☐ niet verlengbaar, a.u.b. retour/*renewal not possible, please return the document*

☐ opmerkingen/*other* :

Briefkaart

FRANKEREN
ALS
BRIEFKAART

Aan de Bibliotheek van de
Technische Universiteit Delft
Postbus 98
2600 MG DELFT

Another process can now enter its critical region for the variable v.

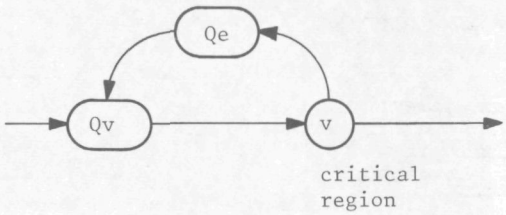


Fig. 2.2. Implementation of critical regions with shared variables and await primitives.

When a process changes v by a statement inside a critical region, it is possible that one or more of the conditions expected by the processes in Q_e will be satisfied. Therefore, after the completion of such a critical region, all processes in the event queue Q_e are transferred to the main queue Q_v. This enables the processes to re-enter their critical regions, to inspect again the shared variable v, and eventually to complete the critical region. A certain amount of busy waiting can be caused by this technique: a process may be transferred several times between Q_e and Q_v before the condition B for which it waits is satisfied.

In the end, the implementation of the constructs presented in this subsection will be based on the existence of a TST instruction and on an interrupt system. The programmer, however, is not concerned with these implementation details. Moreover, it becomes possible for a compiler to check whether the shared variables are used correctly, i.e., that they are operated upon only inside critical regions. This is an aid for ensuring the correctness of the program. The compiler is not capable of performing such a check using semaphores. In this case, the programmer has the complete responsability for using P and V operations in the correct sequence.

The use of the shared variables is illustrated by the example of the producer and consumer processes in subsection 2.1.2. Without providing details, two primitives, "put item" and "get item," are assumed to operate on a buffer. The number of records momentarily present in the buffer is m, and the capacity of the buffer is n, records. The initial value of m is zero, and it is declared to be shared variable in a portion of the system external to the two processes.

With these conventions, the example can be coded as follows:

```
var m: shared integer;
region m do m:=0;

    "Producer"                          "Consumer"
repeat                              repeat
   produce item;
   region m do                         region m do
      await m < n;                        await m > 0;
      put item;                           m:=m-1;
      m:=m+1;                             get item;
  end;                                 end;
end;                                      consume item;
                                    end;
```

The introduction of shared variables makes programming easier and provides an automatic check for correct program constructs.
Still, each program should contain the code for operating on the shared variables. At en even higher level, one can define a shared variable and the operations allowed on it (implemented as procedures), as a separate unit and can make the programs call a procedure of this unit whenever they need the shared variable. In other words, all the critical regions for a shared variable would be collected in one place.
This is the concept of the monitor, introduced by Hoare[11] and Hansen [3].

2.1.4. Monitors.

A monitor defines a shared data structure and all the operations that processes can perform on it. These operations are implemented as monitor procedures. A monitor also defines an initial operation that will be executed when its data structure is created. There is a similarity between the monitor concept and the class concept defined in the programming language Simula 67.

The monitor data can only be accessed through the monitors procedures, which are executed strictly one at a time. Regardless of the number of processes using a set of shared data implemented in a monitor, there is only one copy of the monitor code, to which all process requests are directed.

A monitor can only be initialized once. After initialization, the shared variables and the eventual monitor parameters exist forever.
They are called permanent variables. The eventual parameters and local

variables of a monitor procedure, however, exist only while the procedure is being executed. They are called temporary variables.

A monitor procedure can only access its own temporary and permanent variables, which are not directly accessible by other system components. Other system components can, however, call procedure entries within a monitor.

It is possible to define constants, data types and local procedures within monitors. The local procedures can only be called from other procedures within the same monitor.

A monitor cannot call its own procedure entries (i.e., its monitor procedures).

A monitor declaration will have the following general form [3] ,[11]:

```
type monitorname = monitor (...formal parameters...);
      declaration of the monitor (shared)data;
    procedure entry procname (...formal parameters...);
      begin ...procedure body... end;
    ...declaration of other procedure entries;
    ...declaration of local procedures;
    begin
       initial operation;
    end;
```

Note that the declaration of a monitor procedure is prefixed by procedure entry, while the prefix for a local procedure declaration is simply procedure.

A process calls a monitor procedure by making use of the "dot notation":

```
monitorname . procname;
```

Relative to the monitor implementation, several points should be discussed. First, a solution must be found for dealing with simultaneous calls to the same monitor. This aspect is denoted by Hansen [3] as "short term scheduling" and it arises from the mutual exclusion condition for the execution of the monitor procedures.

A second aspect originates from the requirements of process cooperation: there must be possible to delay a process, inside a monitor, for an indefinite interval of time when a certain condition, expected by the process, is not satisfied and to reactivate the process when an event of another process makes that condition valid. This is "medium term scheduling."

The actual monitor implementation will make use of the simpler concept of semaphore. A semaphore controlling monitor access will take care of the short-term scheduling. It is desirable for the execution time of a monitor procedure to be as short as possible. The processes should not have to wait long before being allowed to enter the monitor.

For process cooperation, the monitor implementation should provide two primitives, delay and continue, and internal queues. When a process is delayed in an internal queue, it will lose its exclusive access to the monitor, which can now be entered by the other processes. However, when a delayed process is reactivated by another process executing a continue in a procedure of the same monitor, it should regain the exclusive use of the monitor immediately, without the possibility of yet another process interfering.

The buffer example of the preceding subsections will be solved with the aid of a monitor. A monitor called "buffer" is defined with two monitor procedures, "put" and "get," operating on a data structure consisting of an array of "pages" (a page is here the memory area necessary to store a buffer item), three integer variables, and two queues. The three integer variables are "head," "tail" and "n," and the two queues are "empty" and "full." The monitor code follows:

```
type buffer = monitor (limit:integer);
  var store:array [.0..n-1.] of page;
      head, tail: 0..n-1; : integer;
      empty, full : queue;
      procedure entry put (x : page);
          begin if n = limit then delay (full);
                store [ tail ] : = x;
                tail:=(tail+1)mod limit;
                n:=n+1;
                continue (empty);
          end;
      procedure entry get (var x : page);
          begin if n=0 then delay (empty);
                x:=store [ head ] ;
                head:=(head+1)mod limit;
                n:=n-1;
```

```
            continue (full);
        end;
    "initial statement"
begin
        n:=0; head:=0; tail:=0;
end;
```

The two processes, producer and consumer, can now be coded:

```
    "producer"                    "consumer"
repeat                        repeat
    produce x;                    buffer.get(x);
    buffer.put(x);                consume x;
end;                          end;
```

In a larger system in which "buffer," "producer" and "consumer" are several components, there will be another component. This component initializes the monitor "buffer"; i.e., the value of the parameter "limit" is specified, space is reserved for the permanent variables "store," "head," "tail," "empty," "full," and "n," and the initial statement is executed.

The simplicity of the process coding is apparent from the completed example. A compiler can easily verify the correctness of single monitor calls. Therefore, once a monitor has been proved to work correctly, a program using that monitor cannot cause an erroneous operation on the shared data. This suggests the usefulnes of the monitor concept in the design of hierarchical systems.

2.2. *Resource allocation and scheduling.*

As it has been shown in the previous section, the limited amount of resources in a computer system is the principal reason for the need of process coordination. This coordination must ensure a correct and efficient sharing of the existing resources between the concurrent processes.

The concept of resource should be self-explanatory. In an very general sense, a resource is any means that a computer system places at the disposal of its users. A computer has physical resources such as a CPU main memory capacity, peripheral devices, etc. It has logical resources such as compilers, program libraries, files, etc. A resource is sharable if it can be used simultaneously by several processes or non-sharable if for a cer-

tain period of time, only one process can use it. Some resources are reusable (e.g., the CPU, the memory space); other are non-reusable (e.g., a punched-card, an instance of a message between two processes). When a resource can be de-allocated, without many difficulties, from one process and allocated to another process, that resource is pre-emptive. A resource which does not have this property is non-pre-emptive (the CPU is a pre-emptive resource, while a line printer is non-pre-emptive).

Resource allocation denotes the actual activity implied in the reservation of a resource for the use of a particular process, while resource scheduling concerns the selection of one process out of the processes waiting for the use of that resource. This scheduling is based on a certain criteria designed to produce an efficient use of the resource. The selection criteria are expressed in scheduling algorithms.

The term resource management can be used to cover the aspects of both resource allocation and resource scheduling.

The mechanisms for process coordination analyzed in Section 2.1. contribute to the correct use of a resource. A supplementary problem in this area is the deadlock. Solutions for it will be presented in this section. Subsequently, several techniques used in resource scheduling will be briefly discussed.

2.2.1. Deadlock.

Deadlock is *"the situation in which one or more processes are blocked forever because of requirements that can never be satisfied"* [13].
The most common situation in which deadlock occurs is that of two or more processes unknowingly waiting for resources that are held by each other and thus are unavailable.
Suppose there are two processes that both need the use of a card reader and of a line printer. Assuming the existence of two monitors, "printer" and "cardreader," each with two procedures, "reserve" and "release," the code of the two processes could contain the following statements:

```
" process 1 "                " process 2 "
      :                            :
cardreader.reserve;          printer.reserve;
      :                            :
printer. reserve;            cardreader.reserve;
```

```
        .                                   .
        .                                   .
cardreader.release;                 printer.release;
printer.release                     cardreader.release;
        .                                   .
        .                                   .
```

The execution of the two processes proceeds asynchronously. Therefore, there is no way of knowing in advance the sequence in which the operations, displayed in the code of the two processes, will be executed. Suppose that process 1 has just executed the operation "cardreader.reserve," while process 2 is executing the portion of code after "printer.reserve" and that both processes continue their execution.

When process 2 arrives at "cardreader.reserve," it will be blocked. The samething will happen with process 1 at "printer.reserve." At this point, both processes will be blocked, waiting for conditions which cannot be satisfied unless an external force intervenes. This is a deadlock.

The problem of deadlock was recognized early in the development of multiprogramming systems, and some ad-hoc procedures have been used to prevent deadlock situations caused by competition for devices.

Generally, the strategies employed in operating systems for dealing with deadlock belong to two categories:

1) - deadlock prevention
2) - deadlock detection and recovery

Holt [13] also mentions the "crash" or "no-strategy" strategy, when an operating system has no provisions concerning a deadlock. If a deadlock occurs, the system will "crash" and a manual recovery procedure must be applied.

A system in which measures are taken to completely avoid deadlock is termed "safe" or "deadlock-free." It uses a strategy for deadlock prevention.

A system is said to be using deadlock detection and recovery when the possibility of deadlock occuring is not removed, but provisions are made for its automatic detection and recovery.

In order to prevent deadlock, a process must satisfy the following constraints when requesting system resources [15]:

- declare in advance all the resources needed;
- before a resource is assigned, execute an algorithm to see if there is a

possible deadlock; if there is not, assign the resource; otherwise the process must wait.

A less restrictive approach [15] to deadlock prevention requires a division of the available resources into numbered classes. A process must request all the resources of a class that it needs at once. Requests for different classes must always be made in the ascending order of the class numbers. By assigning the class numbers carefully, an operating system which uses this approach can offer a greater flexibility than a system in which processes must specify from the beginning all the resources needed.

In a system with deadlock detection and recovery, processes do not have to conform to any constraints when they request resources.
It has been proved that deadlock detection is always possible, but in some cases it may be very difficult, or even impossible, to ensure recovery from a deadlock. One of the algorithms for deadlock detection [15] assigns a number (identifier) to each resource and to each process. It maintains two tables: one indicate the process to which each resource is assigned and the other shows the resources for which each process waits.

This algorithm works as follows.
When a process occupies a resource, it applies a "software lock" that makes the resource unavailable for other processes.
If a process, j, requests a resource k which is already assigned to another process i (i.e. "locked"), with the aid of the tables mentioned above, one can check if the process currently locking the requesting resource is also waiting for some other resource.
If not, process j will simply be put to wait for resource k. If process i is also waiting for a resource, the next step is to determine which process locks that resource. Assume that the resource is locked by a process that is also waiting for a resource. In this case, the previous step is repeated until a process that does not wait for a resource is encountered, when nothing more should be done, or until the initial process j is again encountered, which means that a deadlock has occurred and a recovery procedure must be applied.

The recovery procedure must, basically, ensure that one of the deadlocked processes release one or more resources, while it is "backtracked" or "decomputed" to a point to allocation of those resources. Backtracking is a difficult operation and no general solution has been found yet.

The strategy selected for dealing with deadlock determines to a large

extent the scheduling algorithms for systems resources, thus the organization of the queues required by the different techniques for process coordination. It also determines the selection of the processes to be resumed from these queues.

2.2.2. Examples of scheduling algorithms.

When new user requests (jobs) arrive in a computer system, they are first introduced in an input queue. This enables the jobs to be selected for execution in the order required by the system's policy towards the service of its users.

The portion of the operating system concerned with the selection of the group of jobs that will be taken into consideration for resource scheduling is usually called "job scheduler." It examines the input queue and initiates the scheduling operations. A description of several of the policies used for job selection follows:

1. First come, first served (FCFS): The jobs are selected in the order of their arrival in the system.

2. Shortest processing time first (SPF): Based on an estimation of the required processor time, provided by the user, the job with the shortest estimated time is the first to receive devices and memory.

3. Priority scheduling: At their introduction into the system, the user jobs receive a "start priority," generally calculated on the basis of some data provided by the user regarding the resources that the job needs. The job scheduler will select the job with the highest priority. It is essential to provide the means which the "start priority" of a job increases in time, so that eventually each submitted job will be executed.

Once selected, a job is regarded as a process by the computer. The system allocates resources to this process.

For each type of system resource, different scheduling algorithms have been devised and implemented in the existing operating systems. This subsection contains a review of some of these algorithms, applied to several types of resources: processors, memory, peripheral devices.

a) Processor scheduling. Once a process has obtained all the resources it needs to continue, it is in the "ready" state. The process is able to execute if a CPU is available, or, otherwise, it will wait in a process queue (or "ready queue"). When a processor is released by another process, it executes a "processor scheduling algorithm" in order to determine which

of the ready processes it will next execute. By saving the status of a processor's registers, it can be pre-empted from a process. Therefore, processor pre-emption will not cause deadlock problems. Almost all processor scheduling algorithms make use in some way of pre-emption. A process is not allowed to keep the processor for itself longer than a specified time limit. If the process has not released the processor because of completion, I/O request or other causes, before that time expires, the processor is automatically pre-empted. Another frequent reason for pre-emption is the interrupt system present in almost every modern computer.

Apart form the cases of forced pre-emption, a processor can be scheduled according to one of the following algorithms:

1. Round robin: Each process in the ready queue receives, in turn, a processor for a "time quantum" (usually in the order of 100 ms). If a process consumes its quantum and still needs more processor time, it is placed at the end of the ready queue, so that in the next cycle it will receive another time quantum.

2. Modified round robin: There are several variations, one of which works as follows. When a process enters the ready queue for the first time, it will receive a normal time quantum. If it consumes this quantum and needs more time, the process is introduced in a supplementary ready queue, which will be examined only when the main ready queue is empty. A process in this queue, however, receives a double time quantum.
Several levels of such ready queues can be defined, each with a time quantum twice as long as the quantum granted by the preceding queue [15]. After an I/O operation, a process starts again in the main ready queue.

3. Priority: A process has an assigned or "purchased" priority, and the process with the highest priority in the ready queue is chosen as the next to receive a processor.

4. System balance: In order to keep I/O devices busy, a record of the behavior of each process is maintained, and the "I/O bound" processes will receive a higher priority for the use of an available processor.

b) Memory scheduling. Memory is generally a scarce and quite expensive resource of a computer system. For these reasons memory management has received much attention throughout the evolution of computers.
The least that a system must provide for multiprogramming is a partitioned memory, i.e., the possibility to divide the memory into several areas (partitions) and to allocate each area to a process. Sometimes the number of

partitions and the size of a partition remain constant (e.g., in IBM OS/MFT), but for greater flexibility, both the number of partitions and the size of each partition should be variable (as in IBM OS/MVT).

Before a process can be put in the ready queue (to receive a processor), memory must be allocated for its data and the code of the process brought into the main memory. A technique called overlapping has been developed for the cases where it would be uneconomical or impossible to keep the entire code of a process in the main memory. When the process starts, only the part of the code which is then required is brought into the main memory; later, when some parts of the code are no longer needed, they are overwritten with new portions of the process code. The programmer is completely responsible for the pre-planning of memory overlapping.

In the time-sharing systems, a memory scheduling technique called swapping is often used. A time-sharing process will have a high priority in processor scheduling, but for economic reasons, several such processes share, in time, an area of main memory. When one process has received a certain amount of processor time, its code is written back to secondary memory and the code for another process is brought in the same area of main memory.

An automatic overlapping can be achieved in a virtual memory system. In such a system the amount of memory which can be logically allocated for the set of concurrent processes exceeds the amount of main memory physically present in the system. It has been observed that for relatively long periods of time a process remains inside a relatively small portion of its code (this property is called "program locality"). This means that the process can continue its execution even if only such a portion (its "working set") of code is present in the main memory. If the total size of the all working sets of the ready processes does not exceed the size of the physical memory, the system will work efficiently. It is the responsibility of the operating system to detect modifications and to update the working set of a process.

The code and data which are not part of the current working sets of the processes are stored in the "secondary storage" or "background storage," usually on magnetic drums or magnetic disks.

The virtual memory can be divided into fixed partitions or variable partitions, or each process can have the illusion that the whole virtual memory space is available for itself.

The main memory is also a pre-emptive resource. Therefore, it cannot cause serious deadlock problems. For a virtual memory, however, the operating system must make sure that the aggregate size of the working sets of the ready processes is smaller than the physical memory size. If this is not the case, one or more processes must be blocked, in order to avoid thrashing [15]. The phenomenon of thrashing is produced by very high traffic between the main memory and the background memory. This high traffic is caused by the continual modification of the working sets of the processes, from lack of main memory space. As a result, the background memory devices are artificially overloaded, while the processors remain idle.

c) Peripheral devices. There is a very large variety of peripheral devices used in present computer systems. From the point of view of scheduling, these devices can be classified in three groups:

- dedicated devices,
- shared devices,
- virtual devices.

A dedicated device is a non-pre-emptive resource. Once allocated to a process it will remain with that process until specifically released, when the process does not need it any longer. Examples of devices in this group are card readers, line printers and magnetic tape units. An incorrect scheduling of dedicated devices is very often the cause of a deadlock.

Shared devices are generally the direct access storage devices (DASD) like disks and drums. More processes are allowed to request simultaneously such a device, and the device scheduling is similar in certain respects to that of a processor.

As an example, a technique very often used for moving-arm disks is to collect the requests from the processes in a queue. When a previous request is completed, the position of the disk arm is transmitted to the scheduling algorithm. The algorithm determines the length of the arm movement that each request in the queue would cause and selects the request producing the shortest arm movement. This request is the next to be served by the disk.

There are techniques which convert some normally dedicated devices (e.g., card readers) into shared devices. In a spooling system, for instance, the jobs introduced from a number of card readers are collected into a job input queue on a disk before being processed. The output from the completed jobs is written into a job output queue on the same or other disk before being printed.

A disk used for spooling would, thus, appear to be equivalent to several virtual card readers. Such a technique is especially advantageous when it can be applied to slow peripherals, because it reduces the time that a process must wait for the execution of an I/O request.

2.3. *Information addressing and accessing*

During its execution, a process will make use of several program modules (routines) that must have the possibility of calling each other. While executing a certain routine, a specific set of data is available to the process and a certain number of other routines may be called. These are the access rights of the process at that moment. If the process enters another routine for execution, this may cause a change in the process acces rights.

There may exist routines and data structures which are used in common by several processes, as explained already in Section 2.1.

The information (code and data) that a process may address during its execution forms the address space of the process. The information available at a certain moment, as specified by the access rights, is the execution domain of the process at that time.

The collection of hardware and software devices necessary to provide the right of a process to access a certain piece of information constitutes the protection system of that computer.

The means provided for the actual accessing of information are denoted as information addressing techniques.

The purpose of this subsection is to provide a review of several protection and addressing techniques. These techniques are evaluated with respect to the following two main requirements that they must satisfy:

- Provide means which enable the routine calls and the data accesses required by the processes; and,
- Ensure that no improper actions can occur, i.e., that the routine calls and the data accesssing take place only in a well-specified manner.

Another desirable feature is to allow for the sharing of information between processes in a flexible manner. The merits of the different protection and addressing techniques in this respect are also underlined in the case discussed in this section.

2.3.1. Protection mechanisms in computer systems.

The most elementary mechanism for information protection is provided by the introduction of two modes of processor operation: supervisor-mode and user-mode.

When the computer is in the supervisor mode, it may execute all the instructions in its instruction list without any restrictions. While in the user mode, a group of privileged instructions cannot be directly executed: first a routine of the operating system must be called.

The privileged instructions are intended for the manipulation of sensitive data concerning the system behavior, and therefore only the routines of the operating system, which usually run in supervisor mode, are allowed to access them.

A user program will always run in the user mode. A change from the user mode to the supervisor mode can occur as a result of an interrupt (the interrupts are processed by routines of the operating system) or when a user program explicitly calls a system function by trying to execute a privileged instruction (e.g., the start of an I/O operation).

To avoid interference between user programs running concurrently in a multiprogramming system, the "lock-key" mechanism has been introduced.

It works as follows. The main memory is divided into equal-sized blocks called pages, and a "lock" of usually 4 bits is added to each page. When memory is assigned to a program, all memory pages involved (i.e., all the pages of a memory partition) will receive the same value of the lock. On the other hand, the program, and thus the process which will run it, is provided with a "key" which has the same value as the lock. During execution, at each memory access the key of the program is compared against the lock of the addressed page. Access is permitted only when the key and the lock coincide. Programs of the operating system are assigned a special key (usually zero) which can access the entire memory.

Such a protection can be inadequate sometimes due to the following limitations:

- It does not allow for the sharing of information between processes running in different user programs, unless the shared information is part of the operating system.

- Two modes of operation may not be enough for a flexible organization of an operating system.

Before proceding to the discussion of other protection mechanisms which try to remove the limitations mentioned above, the concept of segment, used as a vehicle for information sharing, is introduced. In the existing systems, this concept has been defined and used with various meanings [1], [16]. For the purpose of the presentation, a segment is the smallest logical unit of information taken into account by an operating system for memory management. Typically, a segment will contain either code or data, but mixed segments may also be used. A code segment may consists of a single routine or several interrelated routines.

A data segment may contain all the date available to a routine in a code segment or only a part of these data (e.g., the elements of an array, or the elements of a row or column of an array, etc.). The size of a segment is generally variable, but it may have lower and/or upper limits.

The existence of segments as logical units for memory allocation suggests the possibility of sharing information among processes at the segment level.

The next protection mechanism that will be analyzed is implemented in the MULTICS system [18],[16],[19]. It is based on protection rings and user access rights to segments of information.

Protection rings are an extension of the supervisor/user mode. The routines of the operating system and those of the user programs are specified as belonging to one of the 8 possible protection rings. Ring 0 has the most powerful access rights and ring 7, the least powerful. On the other hand, the entire information available in the system is divided into segments. For each segment information is provided to specify the rings in which that segment is accessible for read, for write and for execution (Fig.2.3.). This information is maintained in a segment descriptor word (SDW).

The first two rings, 0 and 1, are reserved for the routines and data segments of the operating system. The user programs can be spread over the other 6 protection rings.

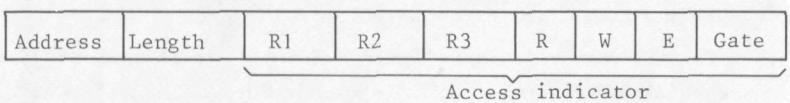


Fig. 2.3. A Segment Descriptor Word (SDW) in Multics.

When a process executes a certain routine, the protection ring of the routine is stored in an extension to the instruction counter (IC) register

(this is called "the execution ring" of the process at that moment). If during execution an address outside the current segment is developed, the descriptor word of the segment that contains that address must be first accessed. The intended sort of access (read, write or execute) is determined by the instruction in which the address was developed. For a write access, if the bit W (write mask) in the SDW (Fig. 2.3.) is 1, (i.e., if writing is at all possible), the execution ring (ER) of the process is compared with the value of R1. The access is permitted only if ER $\leq$ R1. The rings 0 to R1 are called the write bracket of the segment.

Similarly, a read access is allowed if the bit R is 1 when the ER of the process is in the read bracket of the segment, i.e., ER $\leq$ R2.

The execution access is allowed from an execution ring ER if the bit E is 1 and the relation R1 $\leq$ ER $\leq$ R2 is valid. The execution bracket of a segment is thus (R1, R2).

A process is allowed to call a segment as a routine even from outside the execution bracket if the process is currently executing in a ring ER which satisfies the relation

$$R2 + 1 \leq ER \leq R3.$$

In this case, however, the access can only take place through a number of control points or gates, provided immediately at the beginning of the segment. The field "Gate" of an SDW is used to check that the attempted access is not to an address higher than the value specified in the gate field. The rings (R2, R3) are the extension gate of the segment.

In connection with routine call operations, it should be noted that it is relatively easy to implement in hardware [19] a "downward call" i.e., a call from high-numbered rings to lower-numbered rings and the subsequent "upward return." The upward calls and downward returns, however, require software intervention, because they involve the reduction of the access rights. This operation has been considered by MULTICS designers too complex to be completely implemented in hardware.

The protection rings technique offers a greater flexibility than the lock-key mechanism. Specifically it allows a rather straightforward implementation of information sharing. Nevertheless, the implicit "ring property" may sometimes become an impediment, because it allows only a rather restricted number of execution domains with predetermined relations between their access rights. This limitation can be removed in a system based on

capabilities.

A capability is a unique identifier of an information segment[7] . A routine cannot access a segment unless it posesses a capability for that segment. Capabilities are acquired partially when a program is compiled and partially during the execution of a process. The Plessey System 250 will be used as an example of a capability-based system. It is worth noting that this computer does not offer the highest flexibility attainable with capabilities, but it shows a reasonable compromise between flexibility and implementations costs.

The Plessey System 250 [6] divides the information segments into two classes: capability segments, in which the words are interpreted as capabilities for accessing other segments, and data segments, denoting what are usually called code segments and data segments. The processor has a set of eight capability registers, identified as CR0, CR1, ..., CR7. The content of such a register is interpreted always as a capability for a segment. These registers can be operated upon only by a group of capability manipulating instructions and are completely separate from the data registers of the computer. There is no possibility that the data-manipulating instructions will modify the content of a capability register or that a capability-manipulating instruction will operate on a data register. The existence of the Capability Register provides a means for fast access to segments during program execution.

At compilation time, a program is divided into packages. A package consists of a Central Capability Segment (CCS) and a number of satellite segments, which may include other capability segments. The CCS must define at least one code segment and the data structure on which the code segment will operate (Fig. 2.4.). A program package is similar to an execution domain of a process, as defined at the beginning of the section.

While a process is executing in a certain package, two of the capability registers are dedicated, by a hardware convention, to the following tasks:

- Register CR7 contains the capability for the code segment that is being executed. When a "Jump" instruction changes the current code segment to another code segment in the same package, the capability for the new segment is simply loaded into CR7.

- Register CR6 contains the capability for the CCS of the package. The other six capability registers can be used to store capabilities for other segment of the package and, unlike CR7 and CR6, they are accessible to the

programmer.

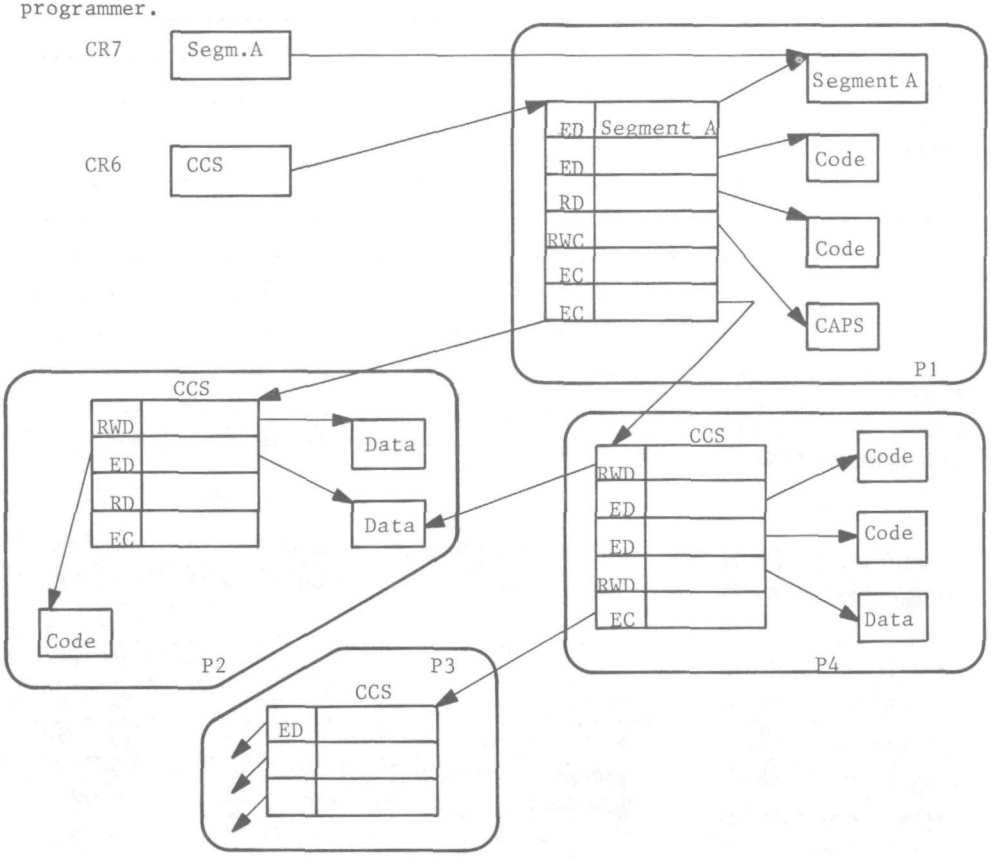


Fig. 2.4. Protection through capabilities in Plessey System 250.

The CCS contains not only the identifier (capability) for each segment, but also information about the kind of access permitted. Thus, the abbreviation ED in Fig. 2.4 means "execute data"; i.e., the corresponding segment is a data segment (in the System 250 sense), and its contents will be interpreted as instructions. Analogously, the meaning of RD, "read data," and RWD, "read write data," should be obvious. RWC in the CCS of the package P1 means "read write capability" and provides access to another capability segment of the package.

The CCS also contains capabilities which allow the routines in the package to call routines in other program packages. One of the most important features of the system originates from the kind of access provided by such capabilities. This type of access is denoted in Fig. 2.4 by EC, "enter

capability."

A CALL statement must have two parameters:

- an Enter-type capability for the CCS of the called package.
- the index in that CCS of the Execute-type capability for the first code segment of the called routine.

The Execute-type capability will be loaded into CR7 at the same time that the Enter-type capability is loaded into CR6. However, before CR6 and CR7 are overwritten, their old values, together with the value of the instruction address register, are preserved in a stack so that these values can subsequently be restored by a Return instruction.

When CR6 is overwritten, the EC access type is automatically changed to a RC, "read capability," which can then be used by the called routine to read the information in the CCS of its own package.

The result of the operations described above is that the execution domain of the process has been changed: the CR6 register points to a new CCS. Thus, all further references are controlled by the capabilities contained in that segment.

There is no possibility for the calling routine to access information in the package of the called routine directly. The EC type of access allows only the passing of the control from one package to another; the packages are completely protected from each other. One can consider the package as the "protection unit" in this system.

A RETURN instruction will restore the contents of the processor registers to the values that they had before the corresponding CALL. Thus, the execution domain is again in the package of the calling routine.

It is important to realize that the sharing of information between program packages is not excluded: segments of code or data can be used in common by several packages (a common data segment is indicated in the packages P2 and P4 of Fig. 2.4). The packages are separated by their central capability segment. These segments specify the access rights to the other segments of the package. It will often happen that the access rights of two packages to a common segment are different.

The organization of the System 250 around capabilities makes a privileged mode of operation for the processors superfluous. Moreover, there is no intrinsic relation between the protection properties of the execution domains (packages), as was the case with the protection rings of MULTICS. In a system based on protection rings, a process, while executing in a highly

protected ring, can access every segment of information available in the less protected rings of process. The capabilities offer a greater flexibility in the sense that a process executing in a certain package has access only to the information strictly required in that package.

The protection properties of a program package make it a suitable device for the implementation of the monitors discussed in subsection 2.1.4. The package organization ensures that its data structure is not directly accessible from outside. A package can be entered only by calling one of its procedures.

2.3.2. Addressing mechanisms.

The usual direct, indirect and indexed addressing modes are not in the scope of this subsection. Instead, the addressing operations necessary to determine the place in main memory of the segment containing the requested address will be considered. In this way, the systems without segmentation are completely excluded from the discussion.

The main problems can then be stated as follows:

1) If the segment is not yet in the main memory, the system must initiate and perform the actions required to bring it there.

2) When the segment is already present in the main memory, the process must receive a "segment descriptor," which indicates the location of the segment.

3) After the access rights of the process have been established by the protection mechanism, the procedures to be followed for routine calls and for data addressing must be specified.

4) The system must provide means for reducing, whenever possible, the time consumed for accessing a segment.

As stated above, these aspects of addressing cannot be completely separated from the protection mechanism used in a certain computer. Actually, in the case of protection through capabilities, the routine calls and the data accessing operations are an integral part of the protection.

The two example systems considered in this subsection are again MULTICS and Plessey System 250.

In MULTICS, the system maintains a table, AST (Active Segment Table), of all the segments currently present in the main memory. For each segment, an entry, identified henceforth as ASTE, contains the segment name, its length, a "Connection list" which identifies the processes currently using the

segment, and several other items. Apart from the AST, each process has its own Process Segment Table (PST), shown in Fig. 2.5. The PST is divided into two parts: a "Descriptor Segment" DS and a "Known Segment Table" KST. A list of several items of an entry in the Process Segment Table follows:

- an indication of whether the segment is in the main memory,
- the address of the segment when it is "active,"
- protection information,
- the symbolic name of the segment.

Theoretically, a process may try to access every segment for which it can provide a valid symbolic name. At the first attempt to access a segment there is no entry in the PST for this segment; therefore a "trap" to the operating system occurs, where the following operations are performed (see Fig.2.5). First, the AST is searched to determine whether the segment is already active (the same copy of the segment will be used by all the processes that need it). If this is the case, the "branch pointer" is copied from the corresponding ASTE and is placed as a new entry into the KST. The process identifier is added to the connection list of the ASTE.

If the segment is not found in the AST, the system must reserve an entry in AST for it. This action may result in the deactivation of another segment if all ASTE have been already reserved. Subsequently, the operating system performs a search of the segment in a hierarchy of directories. Directories are tables used to maintain a record of all the segments that exist in the system. They have a hierarchical organization in order to facilitate the searching. Therefore an entry in such a directory can point either to another directory or to a segment. An entry pointing to a segment is called a branch and contains the information displayed in Fig. 2.5.

When the searched segment is found, some information from the directory branch is copied into ASTE. A switch in the branch is set to indicate that the segment is active. Then, operations identical to those of an active segment are performed.

Now the segment is known to the process, but it cannot yet be referenced. A new attempt to access this segment will activate another system routine which obtains the segment access attributes for the current process (user) from the segment branch and stores them in the corresponding entry of DS. The flag F of the same entry is turned to "ON" to indicate that the segment is present in memory. The next attempt to access the segment can be completed, subject to the protection constraints.

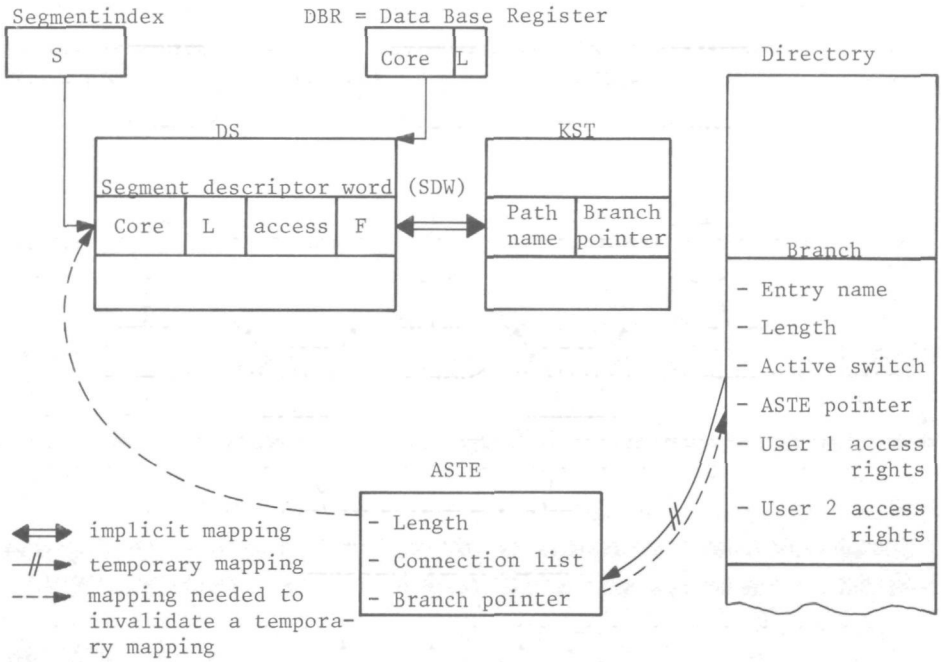


Fig. 2.5. Basic tables used for memory addressing in MULTICS.

In the Plessey System 250, the sharing of the same copy in main memory of a certain segment is achieved also by providing the system with a general segment table. The operations involved are explained with the aid of Fig. 2.6., which illustrates the executions of a "Load Capability" instruction.

The System Capability Table (SCT) provides an entry for each segment in main memory. This entry specifies the memory module where the segment is stored (the field STORE in Fig. 2.6), the address in that module of the word zero of the segment (BASE) and the segment length (LIMIT).

A capability, as it is specified in the CCS, contains an access field and an offset in the SCT, rather than the actual address of the segment.

It should also be noted that the access rights are not given in the entry of the SCT for a segment, but they are indicated separately for each program package. Thus, the access rights to the same segment may be different in different program packages.

An instruction like "Load Capability into CR3 from (CR6)+ offset" is executed in the following way. Using the offset specified in the instruction, the CCS is accessed where the offset in the SCT for the required segment is

obtained. The corresponding entry in the SCT is then accessed, and the information in the fields STORE, BASE and LIMIT is copied into CR3. Finally, the ACCESS rights, as indicated in CCS, are also written into CR3.

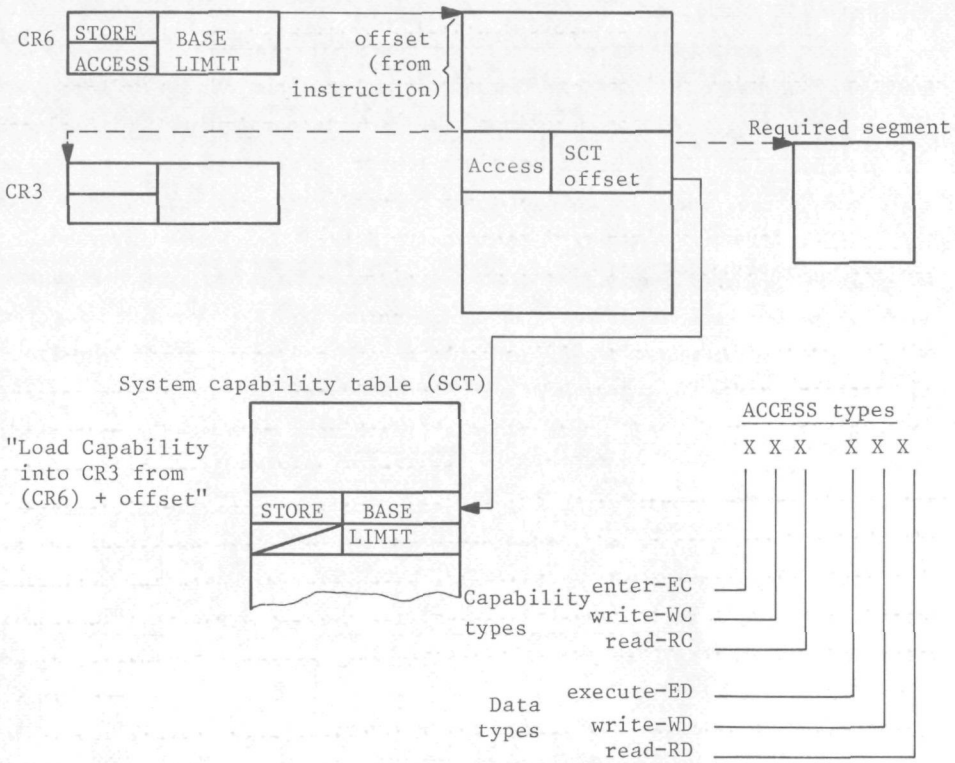


Fig. 2.6. Tables for capability addressing in Plessey System 250.

The programs should not be allowed to access a segment while its position in the main memory is being modified. Therefore, a bit in the capability of the segment indicates that the capability is invalid for the duration of the position modification. An attempt from a program to access the segment in this state leads to a "trap," i.e., a jump to a routine that will passivate the program until the capability is again valid.

During the time that a segment is stored in the secondary memory, a capability for the segment will indicate the segment address in the secondary memory. Therefore, when the segment is brought into the main memory, the operating system must take care that the capabilities for the segment are accordingly modified. A reverse modification should be performed when

the segment is rewritten into the secondary memory.

The operations required for routine calls and for data accessing in a system using capability addressing have been discussed in the previous section. It remains, thus, to examine here the execution of these operations in the MULTICS system.

A code segment is always accompanied by a "link segment" in MULTICS. The code segment has only one copy in the system, whereas there is a corresponding link segment in each process using that code segment. The information in the link segment is used to perform the address mapping required by the fact that there is no restriction in the way that segment numbers are allocated in the Process Segment Table (PST). Therefore, in a code segment conventional addresses will be used for the other segments required. Each process will provide the link segment with the data required for the transformation of the conventional addresses into the corresponding indices in the PST of that process. Details will be given with the aid of Fig. 2.7.

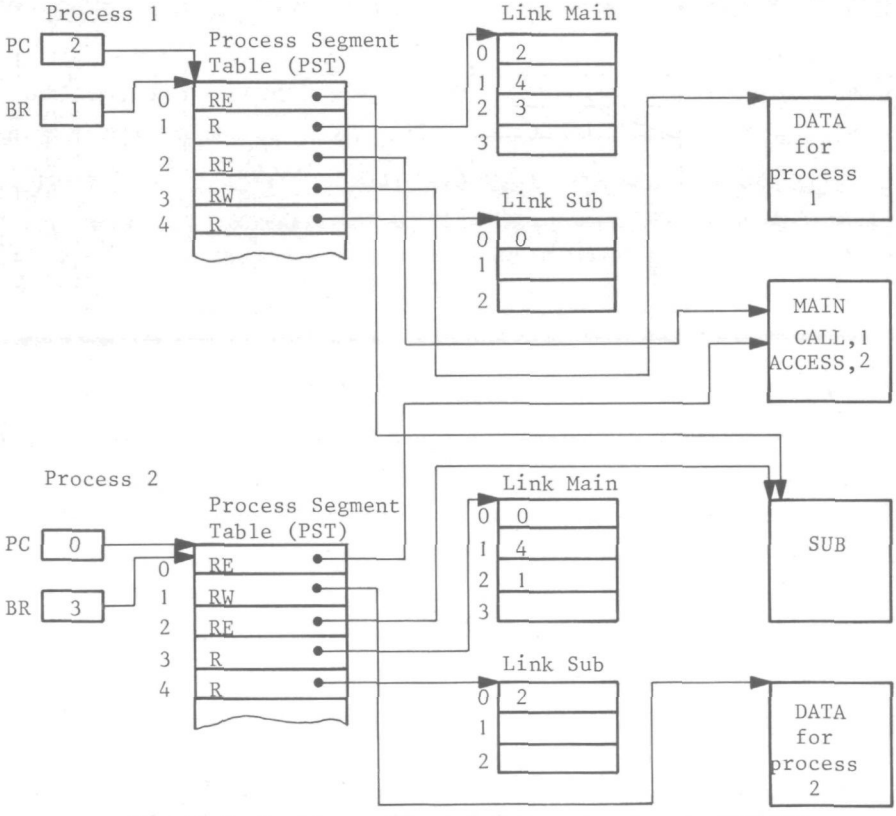


Fig. 2.7. Routine calls and data accessing in MULTICS.

During process execution, the processor must know the index of the current code segment in the PST. This index is specified in the Program Counter (PC) register. The processor must also know the index of the corresponding link segment. The Base Register (BR) provides this information. The two processes shown in Fig. 2.7. share the code for the main program, MAIN, and for a routine, SUB. Each process, however, uses its own data segment.

When process 1 encounters the statement "ACCESS,2" in the segment MAIN, the statement will be interpreted as follows: the process requests access to the data segment with the index 2 in the link segment of MAIN. The link segment can be accessed because BR specifies its index in the PST. The entry with the index 2 of the link segment indicates, in turn, the index in the PST of the required data segment. In this case it is segment 3. In process 2, the same operation would refer to the segment 1 in its PST.

A procedure call "CALL,1" in MAIN leads to the following sequence of operations when executed in process 1. The segment index present in the statement, in this case the index 1, is interpreted again as referring to the link segment of MAIN. There, the index 4 identifies the descriptor in the PST of the link segment of the called procedure, SUB. By convention, the first entry in this link segment defines the index in the PST of the actual code segment for SUB. In this case it is segment 0. The contents of registers PC and BR must be saved in a "process stack segment."
Subsequently PC will be overwritten with the PST index of the new "current segment code." In this case it is 0. BR will be rewritten to point to the new link segment; thus, it will contain 4.

A RETURN statement will restore the old contents of the PC and BR registers, using the values saved in the process stack.

The fourth objective listed at the beginning of this subsection, namely, the reduction of the time necessary to access a segment, is generally achieved by providing a set of high-speed processor registers dedicated to information addressing.

In a computer like the Plessey System 250, the set of 8 capability registers serves, in fact, this purpose: these registers supply immediately the addresses of the most frequently used segments. Thus they avoid the time-consuming operations of accessing the Central Capability Segment and the System Capability Table.

A similar approach is found in MULTICS and in the majority of the systems using segmentation. The addresses of the most recently used segments are

maintained in a set of registers. The manipulation of these registers is completely controlled by the system, whereas in a capability-based system the programmer can access the set of capability registers by means of several instructions.

2.4. Outline of a methodology for operating system design.

In recent years there have been many attempts to define a methodology for operating systems design; examples can be found in [10], [14], [20]. The general picture which emerges from all these works is that the functions of an operating system ought to be hierarchically organized.

At the lowest level of the hierarchy, the system hardware is available, particularly the processors with their machine instructions. Each new level in the hierarchy defines a virtual machine which may use the functions specified in the lower levels and defines, in turn, functions that will be used at higher levels. Thus, the virtual machine defined in the first level of the operating system uses the functions provided in hardware and implements new functions. The virtual machine at level 2 will be built as if the "hardware" at its disposal provides both the functions of the computer hardware and those of the virtual machine at level 1, and so on.

If such a view of the operating system is adopted, a very difficult problem of the design is the definition of the necessary system functions and the selection of their hierarchichal level. Afterwards, decisions should be made concerning the implementation of these functions.

In this context, it is useful to make a distinction between level and module [10]: a level is a set of functions implemented with the aid of functions defined in lower levels, whereas a module comprises some data structures (possible) and a set of functions which share knowledge about a particular design decision (e.g., the details of the data structure of the module). The distinction makes it possible to divide, during implementation, the functions of a certain level into several modules and, on the other hand, to implement in the same module functions from different hierarchical levels.

In the definition and, naturally, in the implementation of the system functions, the available hardware configuration with its specific features and the intented use of the system will have an important role.

References

1. Bensoussan, A., et al,"The MULTICS Virtual Memory: Concepts and Design," CACM, Vol.15, No.5 (May 1972), pp. 308 - 318.

2. Brinch Hansen, P., "Structured Multiprogramming," CACM, Vol.15, No.7 (July 1972), pp. 574 - 578.

3. Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Tr. on S.E., Vol.1, No.2 (June 1975), pp. 199 - 207.

4. Daley, R.C. and Dennis J.B., "Virtual Memory, Processes, and Sharing in MULTICS," CACM, Vol.11, No.5 (May 1968), pp. 306 - 312.

5. Dijkstra, E.W., "The Structure of the THE Multiprogramming System," CACM, Vol.11, No.5 (May 1968), pp. 341 - 346.

6. England, D.M., "Capability Concept Mechanism and Structure in System 250," R.A.I.R.O., Vol.9, B3, (Sept.1975), pp. 5 - 18.

7. Fabry, R.S., "Capability Based Addressing," CACM, Vol.17, No.7 (July 1974), pp. 403 - 412.

8. Haberman, A.N., "Synchronization of Communicating Processes," CACM, Vol.15, No.3 (March 1972), pp. 171 - 176.

9. Haberman, A.N., "Prevention of System Deadlocks," CACM, Vol.12, No.7 (July 1969), pp. 373 - 377, 385.

10. Haberman, A.N., et al., "Modularization and Hierarchy in a Family of Operating Systems," CACM, Vol.19, No.5 (May 1976), pp. 266 - 272.

11. Hoare, C.A.R., "Monitors:An Operating System Structuring Concept," CACM, Vol.17, No.10 (Oct.1974), pp. 549 - 557.

12. Hoare, C.A.R., "Towards a Theory of Parallel Programming," International Seminary on Operating Systems Techniques, Belfast, North Ireland, Aug.-Sept. 1971.

13. Holt, R.C., "Some Deadlock Properties of Computer Systems," ACM Computing Surveys, Vol.4,No.3 (Sept.1972), pp. 179 - 196.

14. Lampson, B.W. and Sturgis, H.E., "Reflections on an Operating System Design," CACM, Vol.19,No.5 (May 1976), pp. 251 - 265.

15. Madnick, S.E. and Donovan, J.J., Operating Systems, Mc Graw Hill, New York, 1974.

16. Organick, E.I., Computer System Organization. The B5700/B6700 Series, Academic Press, 1973, New York.

17. Organick, E.I., The MULTICS System: An Examination of Its Structure and Implementation, M.I.T. Press, Cambridge, Mass.,1972.

18. Saltzer, J.H., "Traffic Control in a Multiplexed Computer System," Ph.D. Thesis, M.I.T., July 1966.

19. Schroeder, M.D. and Saltzer, J.H., "A Hardware Architecture for Implementing Protection Rings," CACM, Vo.15, No.3 (March 1972), pp. 157 - 170.

20. Wulf, W. et al., "HYDRA: The Kernel of a Multiprocessor Operating System," CACM, Vol.17, No.6 (June 1974), pp. 337 - 345.

Chapter 3.

DESIGN OF THE PROCESS COORDINATION PRIMITIVES IN A MULTIPROCESSOR SYSTEM.

The hierarchical approach to operating system design outlined at the end of Chap.2 will serve as a basis for the organization of the remainder of this thesis. The language Concurrent Pascal, defined by P.Brinch Hansen[1], will be used for system implementation.

At the beginning of this chapter, the main design decisions and the peculiar features of Concurrent Pascal are introduced, followed by a discussion of their influence on the system design. Subsequently, the chapter presents the kernel of the operating system, i.e., the virtual machine on which Concurrent Pascal is executed.

3.1. General aspects of the system design.

3.1.1. The main design decisions

The availability of a multiprocessor system is the starting point of the design developed in this thesis. The system must be able to accommodate several identical computational processors (denoted henceforth as CPUs or simply processors) connected to a common main memory which is divided into separately addressable blocks. The input/output operations are performed under the control of a number of I/O processors or channels. A CPU must, however, prepare in advance and store in the main memory the channel

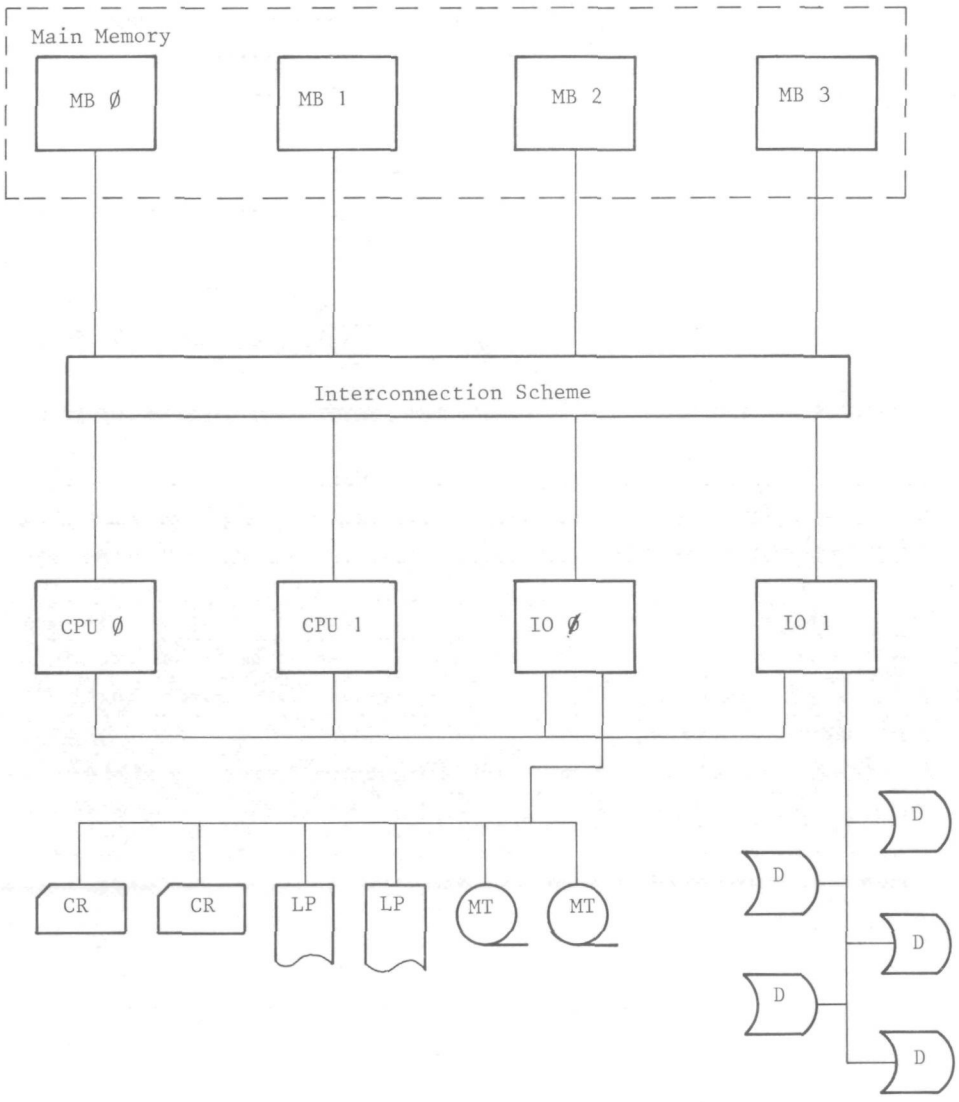


Fig. 3.1. Multiprocessor configuration.

program, and afterwards initialize the channel. A channel program consists of one or more commands, i.e., control words that are recognized by the channels as describing the details of an I/O operation. A channel works independently after it has received form a CPU, during the initialization of an I/O operation, the start address of the channel program. The channel has the possibility of obtaining the commands of that program, one by one, from the main memory. It interprets the commands and sends the necessary control signals to the appropriate I/O device.

A peripheral device is permanently connected to a certain I/O processor. This means that an input/output operation which involves a particular device must always be executed by the I/O processor to which the device is connected.

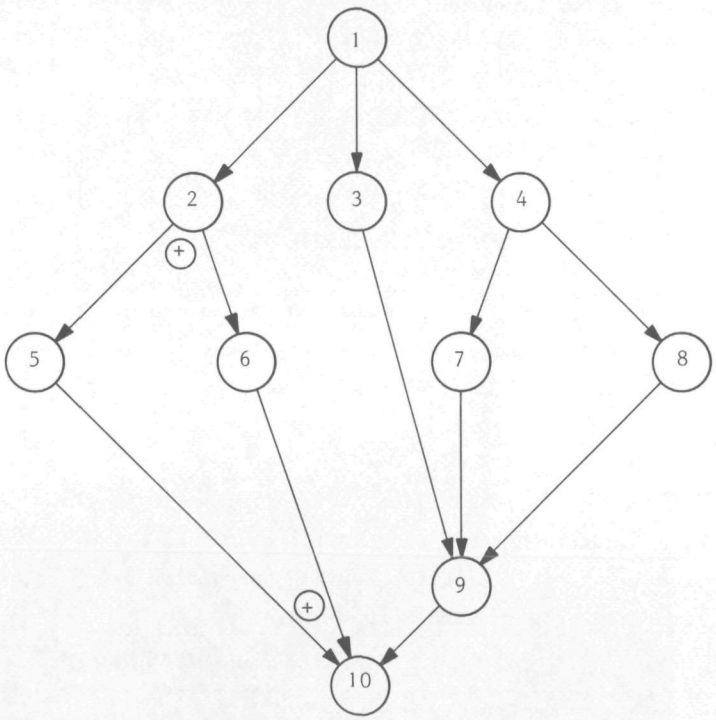


Fig. 3.2. The graph of a parallel program.

The CPUs and the channels of the system communicate with one another by means of a control bus and with the memory by means of a processor-memory

interconnection scheme. The logical design of this interconnection scheme constitutes a major problem in the development of the system hardware. The scheme must ensure an optimum communication between the processors and the memory. The optimum is achieved when the conflict situations between two or more processors trying to access simultaneously the same memory block are reduced to a minimum. However, since the design of an operating system for a multiprocessor is the main topic of this thesis, the particular problems associated with the interconnection scheme are not further pursued.

A possible configuration of the system hardware is illustrated in Fig. 3.1.

Another major design decision refers to the kind of processing provided for user programs. The system offers to its users the possibility of obtaining a short turnaround time for large jobs if the users divide these jobs into smaller units, designated here as tasks, and specify which of the tasks may be executed simultaneously. If system resources are available, such tasks will be indeed processed simultaneously when the program is being executed. Due to this feature, the design system is denoted as a multitasking multiprocessor system. For convenience, a program divided into tasks will be referred to as a parallel program.

The multitasking facility can be usefull for a computation-intensive program where independent computations are discernable, as well as for an I/O-bound program that involves several distinct peripheral devices. The programmer has been made responsible for the separation of a program into tasks rather than providing an automatic recognition of the parallelism for the following reasons:

- The automatic recognition of parallelism in programs (ARPP) is an operation that requires a substantial amount of processing time; the costs involved may be justified for certain production programs, but are prohibitive in a program development system.

- The ARPP is oriented mainly toward discovering the possible parallel computations inside a statement of a higher-level programming language (e.g., a FORTRAN assignment statement). This is not suitable for a multiprocessor, because a computation does not usually require more than one hundred machine instructions, and therefore the overhead involved in frequent switching of a CPU from one computation to another would be of the same order of magnitude as the useful computations.

- The ARPP cannot, at least with known techniques, extract the most

efficient kinds of parallel operations in a program written in the existing higher level programming languages.

It has been suggested [2] that in a multiprocessor system the length of a task should be of at least 50 statements of a higher level programming language. This will make the time consumed in processor switching negligible in relation to the time spent for useful computations.

Figure 3.2. shows the graph of a parallel program; a task is represented as a numbered circle and the ordering of the tasks is indicated with the aid of arrows and of the sign $\oplus$.

The program graph should be interpreted as follows. When task 1 is completed, tasks 2, 3 and 4 may start simultaneously. At the completion of task 2, task 5 or task 6 will be selected, based on a certain result obtained in task 2. When task 4 is completed, tasks 7 and 8 may start simultaneously. In order to start task 9, three tasks, namely 3, 7 and 8, should have been completed. The last task of the program, 10, may start when task 9 and the task selected out of tasks 5 and 6 are completed.

From the program graph presented in Fig. 3.2., it is apparent that, at different points of the program execution, there are tasks which can be processed simultaneously provided processors are available. The design developed in this work allows each task to act as an independent program in a process dedicated to the execution of user programs.

Finally, it has been decided to use the language Concurrent Pascal for system implementation. This language (see Appendix 1) has the following advantages:

- It offers a very general method, namely monitors, for the description of inter-process communications.
- It allows the verification of the correctness of most of the operations at compile time, simplifying the system testing.
- It permits a hierachical implementation of the system and prevents new components from destroying those already tested. Old components cannot call new ones, and the new ones can only call the old ones through routines that have already been tested.

The fact that the language admits only a fixed configuration of the operating system can be considered to be a disadvantage. The language requires a new compilation of the system whenever the configuration must be changed. However, since configuration changes do not occur frequently this disadvantage is not very important.

3.1.2. The influence of the language Concurrent Pascal on the structure of an operating system.

Concurrent Pascal is a language designed for the implementation of operating systems. Its main feature [1] is the introduction of several abstract data types considered to be useful in structuring a system. An abstract data type is defined by Brinch Hansen [1] as "the combination of a data structure and the operations used to access it." The operations mentioned in this definition are implemented as routines that can be called by other system components. Except for calls to these routines, there is no other possibility to act upon a component declared to belong to an abstract data type.

An operating system appears as a Concurrent Pascal program constructed from components of the following abstract data types: processes, monitors and classes. Processes are the active components of the system in the sense that they control the execution of the sequential programs submitted by users or system programmers. Processes synchronize and exchange data with each other by means of monitors and access their own data using classes.

The introduction of abstract data types has important consequences for the reliability of a Concurrent Pascal program. An appropriately written compiler can verify the correctness of the operations on abstract data components before the program is put into operation. In other words, a compiler can ensure that such components are accessed only in the manner permitted by their definition. Consider again the monitor that implements a buffer, defined in Section 2.1.4. It is relatively easy to verify during the compilation that a program using the buffer will do so only by calling the monitor routines "put" and "get." The buffer data are, therefore, protected from erroneous operations, a fact which contributes to an increase in program reliability.

The verifications at compile time of the programming relationships that remain unchanged for long periods of time is a feature permanently pursued in the definition of Concurrent Pascal. This approach to system design produces programs whose interactions with each other are practically correct even before testing, but one must be aware that there are also certain inconveniences. The very strict type checking makes it necessary, for instance, to duplicate definitions which are otherwise almost identical: a buffer for the transmission of memory pages of 1024 bytes and a buffer for pages of 256 bytes must be declared as distinct types, although they

perform the same operations.

The number of processes and the function of each process in an operating system is determined by the system configuration. In principle, a number of processes direct the execution of user programs, while other processes control the operation of I/O devices. Once the system has been initialized, the number of processes, monitors, and classes remains constant until another initialization takes place.

A graphical representation of the definition of an process is given in Ref. [1] and reproduced here as Fig. 3.3. The access rights of a process specify the type of monitors that the process may call in order to cooperate with other processes. The private data of the process are either classes or variables and constants of other standard types. A sequential program describes the activity of the process. When the operating system is initialized, the access rights of each process are established (i.e., the actual monitors are indicated) and the sequential programs of all system processes are activated.

Fig. 3.3. Process definition

An essential requirement provided for by Concurrent Pascal is the possibility of a sequential program calling other sequential programs and transmitting parameters during these calls. Suppose that a process has the function of executing user programs in a single-user system. Initially, this process will execute a program which can accept and interpret commands from the user. If a compilation command arrives, the initial program must be able to call the appropriate compiler and to indicate to the compiler the place where the source code is found. During compilation, it will be necessary to create temporary files by calling another program: "file." A process has a component called "program stack" which implements the operations required by program calls.

The interaction between the sequential programs and the operating system takes place through interface procedures defined in the operating system.

In accordance with the principles of Concurrent Pascal, a compiler must check that the programs do not attempt to interact with the system other than through the permitted interface procedures. Besides, not all the programs necessarily have identical rights in calling system procedures.

Before starting the translation of a program, a compiler must have, therefore, a list of the interface procedures available to that program. The list is called the program prefix and contains the names of the interface routines, together with the parameter types of each routine. A program is rejected if it tries to call the system for an operation that is not specified in the program prefix. There is no supplementary access rights checking for routine calls during program execution.

Separate prefixes are provided for the different categories of programs. The system determines the category of a particular program according to a set of criteria defined by the system designer. The compiler receives information about the category of a program and must be able to select the appropriate prefix.

The implementation of the interface procedures is the responsibility of the system processes. It should be noted that not all processes execute complex programs, and therefore there are processes which are not provided with separate interface procedures.

An interface procedure transforms the calls presented in a general form by a program into specific calls to the proper routines of the monitors that can be accessed from that process.

The structure and the role of monitors have been discussed in Sec.2.1.4. In the definition of a process, the monitors required for the coordination with other processes are specified as formal parameters. During system initialization, the processes and the monitors are declared as variables of an initial process. In this way it becomes possible to specify the actual parameters of each process. The hierarchical organization of the system demands that the monitors are also provided with parameters and that they can call each other. These requirements produce nested monitor calls and influence the internal organization of the monitors. A process that has entered a monitor in an outer level of the hierarchy maintains its exclusive access to that monitor until the request passed to the monitors of the inner levels is satisfied. As a result a process whose request entails a sequence of hierarchical monitor calls can be delayed only in a queue of the last monitor to which the request was passed.

Classes are introduced as abstract data types that describe components private to a certain process. Classes simplify the process code, providing routines by which the process can operate on its own data.
This means that the programmer need not be concerned with the details of

the respective data structure every time an operation on that data is required.

Classes may also have parameters which are monitors or other classes. A class may, in turn, be a monitor parameter but, by definition, it cannot be a process parameter. Due to the definition of a class abstract data type, a class component is always declared as a variable of a certain process and is initialized by that process.

The implementation of the class does not create particular problems. However, the virtual machine executing a Concurrent Pascal program must possessthe set of primitive operations required by monitors.

The entire system appears as a hierarchy of functional levels, as represented schematically in Fig. 3.4.

At the lowest level in the hierarchy one finds the system hardware. Above this level, a virtual machine which supports the primitive operations required in the implementation of monitors is created. The set of procedures written for this purpose is designated as the kernel of the operating system. Once the kernel exists, the remainder of the operating system can be written as if running on a machine that has, in its instruction set, operators for entering and leaving monitors and for delaying and continuing processes in monitor queues. The number of CPUs is irrelevant above this level.

The portion designated as "operating system" in Fig. 3.4. contains, in fact, in a real system, several hierarchical levels implemented as a set of monitors which can call one another. At the outer level of the operating system, processes realize the interface with the utility and user programs.

At this point a few words ought to be said about the programming languages used in writing the kernel routines and the user programs. The kernel routines must implement operations that do not exists in Concurrent Pascal. In principle it is possible to write these routines in a higher level language, e.g., sequential Pascal, and let the compiler translate them in machine language.

However, in practical situations the kernel will be written most likely in the assembler language of the machine, mainly for reasons of efficiency. On the other hand, the system must be prepared to accept user programs written in any programming language, provided that the compiler of that language respects the conventions of the program-process interface.

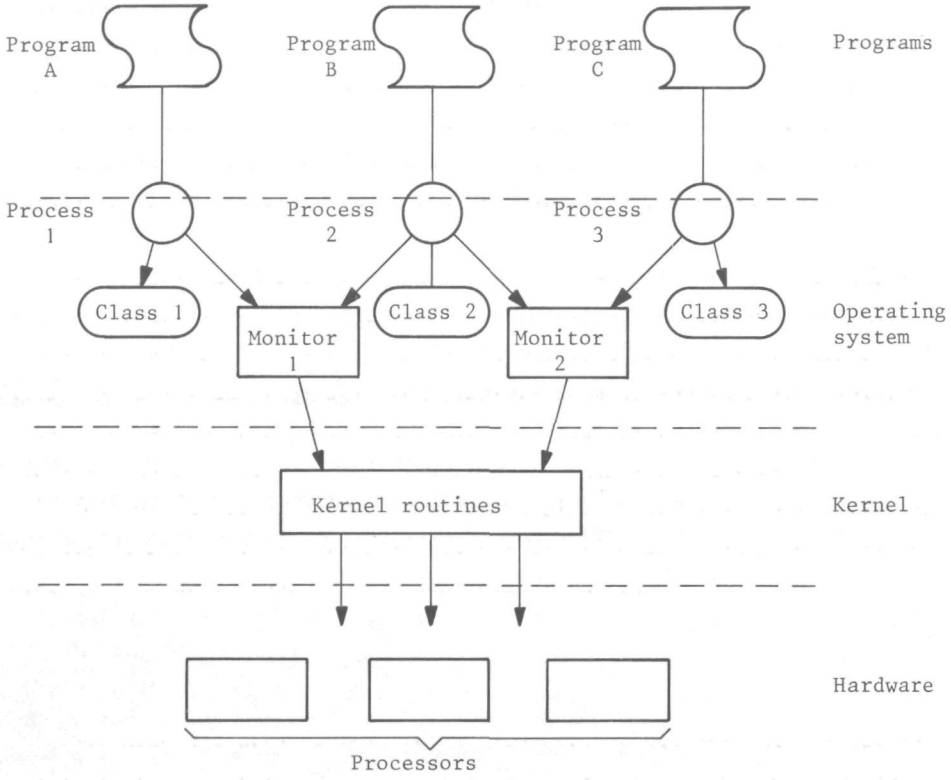


Fig. 3.4. Schematic representation of the system hierarchical levels.

3.2. *Kernel functional design*

The role of the kernel is to create a virtual machine which appears, to an operating system written in Concurrent Pascal, to be equipped with a number of processors equal to the number of processes in the system and to be able to perform the scheduling operations required by the monitors. Thus, the kernel must implement, with possible aid from the machine hardware, the allocation of the CPUs to the processes ready to execute. On the other hand, the kernel must provide the routines for the primitive operations required by monitors.

The kernel routines will appear as operating at two different levels. The monitor primitives have the possibility of calling explicitly the

routines dedicated to processor allocation.

This section contains a functional description of the activities that should take place in the kernel. A subsequent section will be dedicated to the detailed description of the implementation of the kernel.

3.2.1. Analysis of the monitor primitive operations

According to the monitor description, there are four primitive operations required in the implementation of the monitor concept:

- enter a monitor;
- leave a monitor;
- delay a process in a monitor queue;
- continue a process that is waiting in a monitor queue.

Each of these operations may produce modifications in the status of the process executing the operation and/or in the status of other processes in the system. The system provides a data structure that always contains the relevant information about a process.
Important items (process attributes) in this data structure are the contents of the processor registers at the moment when the process was blocked, the execution priority, the resources allocated to the process, etc. Such a data structure is further denoted as a Process Control Block (PCB). The changes in the status of a process are reflected in the modification of the contents of its PCB.

It has already been stated that the access to a monitor is controlled by a semaphore. From the definition of the semaphores it results that the kernel must have a queue of PCBs as one of its data structures. Another reason for providing queues of PCBs in the kernel is the probability that the number of processes ready to execute at a certain moment is larger than the number of system processors. In such a case, the available processors will be allocated to the ready processes having the highest priority. The other processes must continue to wait in a queue of ready processes, until another processor becomes available.

Assuming that the system has more than one processor, it may occur that two or more processors will try to access a certain queue at the same time. Therefore, it is essential to protect each queue with a lock against inadvertent access. This is a fundamental requirement in a multiprocessor system. It may bring about cases in which "busy waiting" must be accepted.

The implementation of the queue lock requires that the CPUs are able to execute a TST instruction (see Sec. 2.1.1.). In this situation, a lock will be a memory location associated with the queue. A request for access to the queue must always start with a TST instruction on that memory location. The access to the queue is permitted if the lock is "open" and denied otherwise. The processor that has been denied access to a queue remains in a test loop until the access is granted. The last operation executed during the access to a queue must reset the lock to "open ."Another processor may now pass the lock test.

Provided that PCBs and PCB queues exist, the monitor primitives can be described as follows.

The primitives "enter a monitor" and " leave a monitor" are in fact, identical with the P and V operations on a semaphore. The designer must provide a semaphore for each monitor of the system. A request from a certain process to enter a monitor is always directed to the corresponding semaphore, where a P operation is performed. Thus, if there is no other process operating in the monitor, i.e., if the value of the semaphore is 1 (or "true"), the process is allowed to continue and the semaphore becomes zero ("false"). Otherwise, the PCB of the process is added to the semaphore queue, the process status information is saved, and the processor must be reallocated.

When a process leaves a monitor, it performs a V or "leave" operation on the semaphore of that monitor. Thus, the process examines first the semaphore queue. If it is empty, the semaphore value is restored to 1 and the process continues. Otherwise, the first process is taken out of the semaphore queue and its PCB is introduced in the queue of ready processes. One of the attributes of the process, its priority in relation with the other processes, will determine whether a reallocation of the processors should take place at this moment.

The "delay" and "continue" primitives are more complex. A monitor can be provided with internal queues, i.e., data structures that can hold a reference to, at most, one process at a time . Note that these internal queues are not related in any way to the kernel queues introduced above. When a process must be delayed in a monitor because a certain expected condition has not been satisfied, the address of its PCB is stored in the associated internal monitor queue. The status information of the process is saved in the PCB, and afterwards the queue associated with the semaphore of

the monitor is examined. If the queue is not empty, the first process in the queue is added to the queue of ready processes; otherwise, the semaphore is set "free". A processor reallocation takes always place after a delay operation, because the processor is preempted from the delayed process.

The operation "continue" consists of the following actions. The monitor queue that is indicated as a parameter of "continue" is examined. If it is found empty the operation is identical to "leave." Otherwise, the process from the monitor queue is introduced in the queue of ready processes and the process that has issued the "continue" leaves the monitor, without altering the value of the monitor's semaphore. The process that had been delayed, whose control point remained in one of the monitor's routines, can therefore continue its operation without interference from other processes trying to access the same monitor).

A flowchart representation of the four monitor primitives is given in Fig.3.5. The operation "reallocate" in the "leave monitor" and "continue" primitives is indicated with a dotted line in order ro suggest that this operation is actually performed only when process priority conditions require it. The details of introducing a PCB in and taking a PCB out of a kernel queue and the details of the reallocate operation will be examined in the section dedicated to the kernel implumentation. At the level of functional description, these functions may be considered as elementary operations.

A separate copy of the routines enter, leave, delay and continue may be provided for each monitor. However, such a solution would require a large amount of memory space, because of the large number of monitors in a complex system. A better solution is, therefore, to write the routines as re-entrant code and to use a single copy of this code for all the monitors.

The re-entrant code will allow a processor to start the execution of a routine before another processor completes an earlier-started execution of the same routine. In this way, the execution speed is not substantially affected and the memory space is more efficiently used. Obviously, there is a separate semaphore with the additional queue for each monitor. Pointers to these variables must be parameters in a call to one of the four routines.

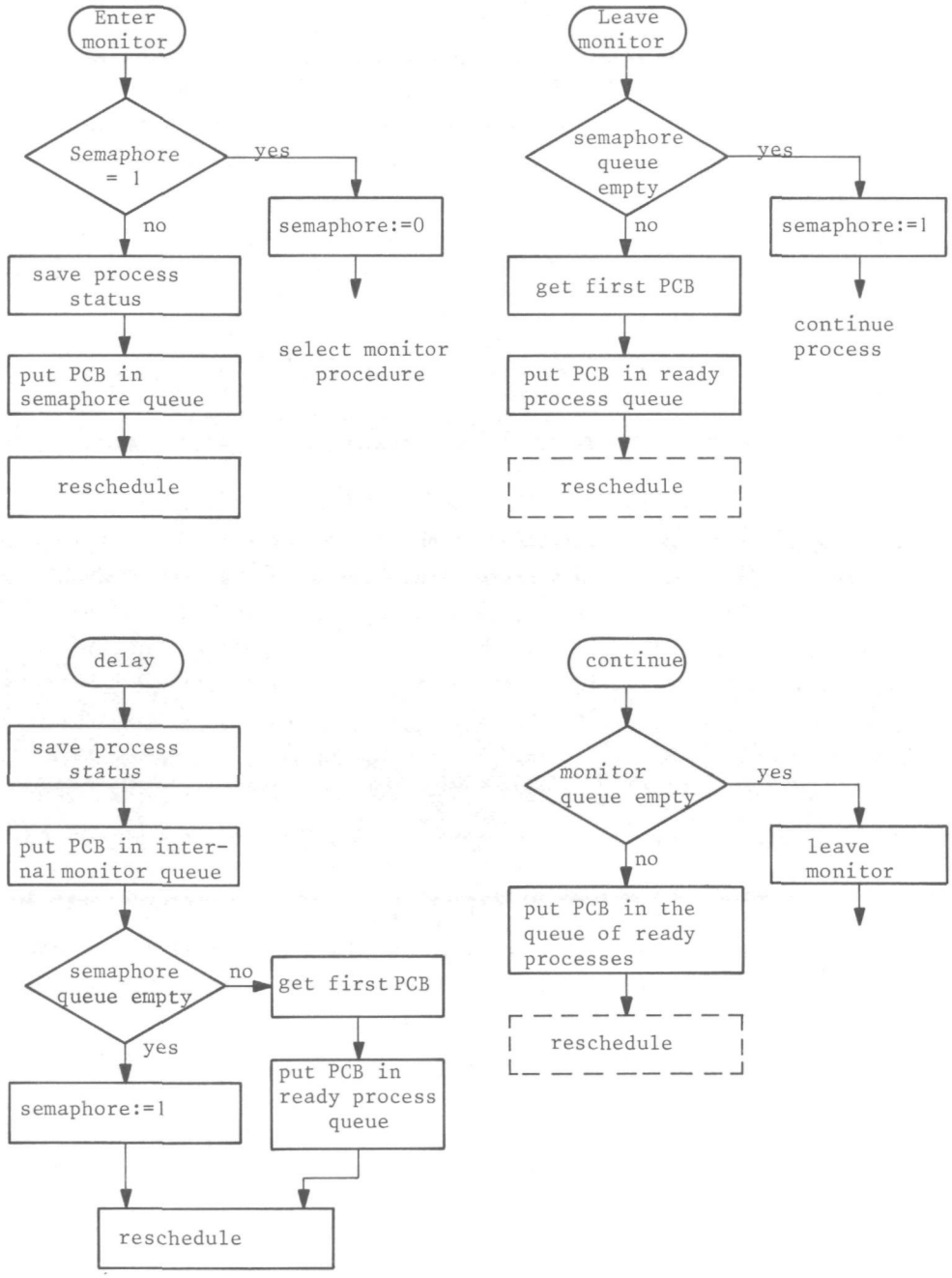


Fig. 3.5. Flowcharts of the monitor primitives

3.2.2. The relation between the kernel and the interrupt system.

It has been stated in Sec. 3.1.2. that the activity of each peripheral device is controlled by a process, identified in the following discussion as the driver. Since all the I/O operations are buffered, when a process needs to transfer data to or from a certain device, it operates on the corresponding buffer. The buffer is implemented as a monitor, and the monitor routines take care of the coordination between the calling process and the driver. Usually, the driver process is written as a cyclically operating piece of code. After having performed an I/O operation it waits in a monitor queue until a monitor routine determines that an I/O operation must be performed again. In this situation, the driver is reactivated (by a "continue" operation). When it obtains a CPU, the driver prepares the I/O operation; i.e., it builds a channel program and sends the address of this program to the appropriate channel (I/O processor). Afterwards, the channel will start the execution of the I/O operation, while the driver enters a special monitor and waits there until the completion of the I/O transfer. When the transfer has been completed, the channel must have the possibility to reactivate the driver. The driver will eventually reactivate the process that has requested the I/O operation, and then it will delay itself in the buffer, completing its operation cycle.

The system's drivers may differ in detail from the one in the example sketched above, but the principle of all the drivers remains the same. The driver executes on a CPU in order to prepare an I/O operation. Then the actual execution of the I/O operation is controlled by an I/O processor.

When the I/O operation has been completed, the driver will again need a CPU to record the effect of I/O completion and to try to obtain further work.

In a system organized as the one illustrated in Fig. 3.1., there is no possibility for an I/O processor to access directly the queue of ready processes in the kernel. Therefore, the I/O processor places a signal on the Control Bus, and the logic of the Control Bus must ensure that eventually a CPU will be required to observe this signal.

An interrupt system is provided for this purpose. It is usual to define several priority levels for the interrupt signals, and likewise a priority level is always specified for a program running on a CPU. The priority level of a program is stored in the Processor Status Word (PSW).

An interrupt signal can be taken into consideration only by a processor executing a program at a priority level lower than the priority of the signal. When an interrupt signal is acknowledged, an interrupt routine must be performed in order to examine the cause of the interrupt and to take the necessary actions. The interrupt routine is a piece of code, and therefore it should also have a certain priority, but this priority should not necessarily be the same as the interrupt signal that activated it.

The approach taken in this work regarding the treatment of interrupts is to consider an interrupt signal as being a request to enter the monitor where the driver has delayed itself after starting an I/O operation. A routine in that monitor will execute a "continue" operation to place the driver again in the queue of ready processes.

3.3. Kernel implementation.

This section is dedicated to the detailed description of the kernel procedures and to the examination of possible hardware aids for the kernel operations. The kernel procedures are described in Concurrent Pascal. This is not necessarily the language in which the routines will be written for an actual system. Indeed, Concurrent Pascal does not provide a construct equivalent to a TST instruction, required in several occasions. Some other "non-standard" constructs are introduced to express a few peculiar operations required in the kernel. The selection of Concurrent Pascal was determined by its features concerning the representation of data structures and by the desirability of using a single programming language for all the examples contained in the thesis.

3.3.1. The implementation of a kernel queue.

A queue of PCBs has been indicated in Sec. 3.2.1. as a basic element in the organization of the kernel. Formally, the queue can be defined as a data structure consisting of two address pointers and a Boolean variable, "lock", together with the operations "testlock," empty," "get" and "put". The two address pointers serve to indicate the location of, respectively, the first and the last element in the queue. A queue element (a PCB), thus, must contain in turn two pointers that allow its chaining in a certain queue. The "lock" indicates whether the queue is "free" (i.e., it can be operated upon) or not. Every access to the queue must start with testing the lock value, i.e., with the operation "testlock." A new element is introduced in the

queue as the last element with the aid of the procedure "put." The function "get" gives the pointer to the first (the longest waiting) element in the queue. Finally, "empty" determines whether there are any elements in the queue at the moment of the call.

The queue data type is presented below as a Concurrent Pascal monitor.

```
type queuetype = monitor;
    var succ, pred: @queuetype;
        lock: boolean;
        procedure testlock;
            begin l : if lock = false then goto l;
                lock : = false;
            end;
        procedure entry put (newelem :@ queuetype);
            var last:@ queuetype;
            begin testlock;
                last := pred;
                pred := newelem;
                newelem. pred := last;
                newelem. succ := last.succ;
                last.succ := newelem;
                lock := true;
            end;
        function entry get :@ queuetype;
            var first, second :@ queuetype;
            begin testlock;
                  get := nil;
                  if succ <>this queuetype then
                  begin first := succ;
                        second := first. succ;
                         succ  := second;
                        second.pred := first.pred;
                         get := first;
                  end;
                  lock := true;
            end;
```

```
function entry empty : boolean;
    begin testlock;
        empty := succ = this queuetype;
        lock := true;
    end;
"initialization"
begin
    succ := this queuetype;
    pred := this queuetype;
end;
```

There is indeed a resemblance between a queue and a monitor. In both cases simultaneous requests to enter the data structure can exist, but the operations on the data structure exclude each other in time. One should be aware, however, that the scheduling of the simultaneous requests is not the same for a queue and for a monitor.

Each queue must be represented by its own queue variables (i.e., the two pointers and the lock), but there is only one copy of the code for the queue operations, written as re-entrant procedures.

The initialization of a queue data structure is an operation executed at the request of the kernel component where the queue is used. The request will appear during the initialization of that component at system loading.

The procedure "testlock" is actually implemented with the aid of a TST instruction, but this fact cannot be expressed in Concurrent Pascal.

3.3.2. Processor allocation; the queue of ready processes.

The interface between the computer hardware (more precisely, the CPUs) and the operating system is implemented by the kernel routines dedicated to processor allocation. These routines accept as their input data the collection of processes ready to execute (provided by the routines implementing the monitor primitives) and information about the status of the CPUs (provided by the hardware). Based on these data, the routines allocate processors to the ready processes, in priority order. In other words, in a system with N processors, if the number of all processes ready to execute at a certain moment is higher than N, the first N processes in descending

priority order will be actually running.

The collection of processes ready to execute is in fact organized as a group of kernel queues, one queue for each process priority level. For the sake of brevity, the set of queues is further referred to as "the queue of ready processes." Note, however, that this name designates more than one queue. The basic priority of a process, i.e., its priority when not requesting a system function, is established in the operating system and is recorded in the PCB of the process. The kernel can modify temporarily the effective priority of a process, e.g. when a process is executing in a monitor or when a process prepares or examines the result of an I/O operation. The kernel routines can easily identify a situation when the process priority must be increased. Due to the possibility of nested monitor calls, a complication occurs however in determining whether the process must return to its basic priority. This problem is solved by providing an entry into the PCB, "nesting," which indicates the level of nested monitor calls. The value of "nesting" is increased by one whenever the process enters a monitor and is decreased by one when the process leaves a monitor. When the "nesting" is strictly positive, the process has a higher priority; otherwise it has the basic priority.

The kernel provides a routine which introduces a ready process in the corresponding queue. This routine can be called during the execution of one of the monitor primitives.

Another situation which may occur in the system is the following. A process releases its processor (e.g., after a "delay" operation), but there are no other ready processes. Temporarily, the processor becomes idle. It is desirable that in such a case the processor does not execute instruction fetch cycles. Thus, it does not access the main memory. This will enable the active processors to access the memory with less contention delay. A speciall instruction, WAIT, is provided in the instruction list of the CPUs for this purpose.

The instruction has the following effect. The processor priority is reduced to the lowest level, and the instruction fetch cycle is stopped. When the next interrupt signal occurs, the idle processor will be the one selected to acknowledge the signal and to leave the idle state.

The operation of process preemption and processor allocation to a process with a higher priority, mentioned above requires a certain kind of communication between the CPUs which is realized by using the Control Bus.

A predominantly software-implemented solution for this problem is possibly the following. In addition to the PCBs, the kernel maintains information about the status of each processor in the form of Processor Status Blocks (PSBs). When a new process is introduced in the queue of ready processes, the PSBs are examined. If a processor is found running at a priority level that is lower than the priority of the new ready process, a request is placed on the Control Bus addressing that processor. Such a request is equivalent to an interupt signal and will be acknowledged by the addressed processor at the end of its current instruction. The appropriate kernel routine is selected and performs the "switching" of the processor from one process to another.

In the present work, however, the hardware of the Control Bus has been extended in order to achieve a faster process switching. The Control Bus organization for this solution is represented in Fig. 3.6. The scheme contains a Bus Master connected to the system processors via three buses: an interrupt request bus, a current priority bus and a master request bus.

The interrupt request bus is used by the CPUs and the I/O processors to place an interrupt signal. A signal from a CPU is raised whenever that CPU introduces a process in the queue of ready process.

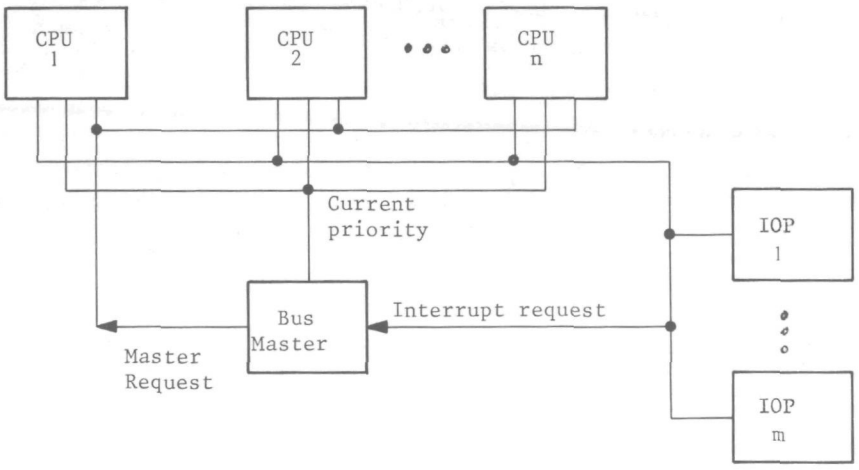


Fig. 3.6. Logic structure for the Control Bus.

The current priority bus is connected only to the system's CPUs. A CPU can place on this bus the priority of the process that it is currently running, the CPU address (identifier) and a signal to request the attention of the Bus Master.

The master request bus is the output of the Bus Master. When a processor (CPU) arrives at an interruptible point (in principle after each instruction), it examines the master request bus. A line on this bus indicates whether a request is pending, while the other lines carry the address of the processor to which the request is directed and information about the nature of the request.

The Bus Master continuously examines the signals on the interrupt request bus and compares their priority with the priority of the CPUs. If a CPU with a priority lower than that of an interrupt signal is found, the Bus Master places a signal on the master request bus and waits until this signal is acknowledged.

Whenever a CPU changes its priority during the execution of a process (e.g., when it leaves a monitor and returns to the basic priority of the process or when it enters a monitor and raises its priority level), it must inform the Bus Master about the modification. A special instruction has been provided for this purpose.

The effect of this instruction is now described.The new priority level, together with an "attention signal" and the processor address are placed on the current priority bus. Subsequently, the operation of the processor is continued only after it receives, over the same bus, an acknowledgement from the Bus Master that the request has been honored. The Bus Master accepts the request and updates the information it maintains about the processor's priority.

A processor does not acknowledge requests on the master request bus if they have priority level lower than that of the processor. Therefore, if the current request concerns the processor that has just modified its priority, the Bus Master should compare the new priority level with the priority level of the request. If the latter is lower, the request must be re-computed, to reflect the new system status.

Another problem that should be solved in a detailed design of the Bus Master arises from the existence of a finite time between the acknowledgement by a CPU of a request on the master request bus and the completion of the process switching operation.

Assume that the request has been to take a process from the ready processes queue and that the process has been the only one on its priority level. In this case the signal on the corresponding line of the interrupt request bus must be dropped. This should happen before the Bus Master takes a new decision concerning the same interrupt level. Otherwise the Bus Master might decide to take again the same process from the ready processes queue.

A possible solution to this problem is based on the temporary invalidation of the lines of the Interrupt Request bus. Thus, after accepting a request for process switching and before sending an acknowledge signal to the Bus Master, a processor invalidates the interrupt level of the request. In this way, the next decision of the Bus Master can be made, without taking into consideration the invalidated level. Alternatively, the Bus Master can be so designed that it waits until the interrupt level is revalidated. This will happen later in the routine that accomplishes the process switching.

It is required that a Bus Master implementation provides a circuit which indicates, before each new decision, the processor having the lowest priority and the value of that priority. If there are several such processors, the circuit indicates the first processor encountered. Assuming that this circuit has a serial operation, the examination of the priority levels of the processors may always start with the same processor. However, a more uniform processor utilization is achieved if the examination is done in a round-robin fashion, i.e., starting a new cycle from the processor that follows, in a conventional numbering, the one used as the starting point in the previous cycle.

The round-robin examination is also useful for the processing of the clock interrupt. If a System Clock is provided as a peripheral device, it is easy to direct each clock interrupt to another processor, by using the round-robin mechanism.

With the newly introduced hardware features, the routines that operate on the queue of ready processes can be written as follows:

```
const maxpriority = 15;
      maxreg = 16;
```

```
type   PCB = record
               registerarea : array [. 1..maxreg.] of integer;
               succ,pred    : @ queuetype;
               callparam    : array [. 1..4.]  of integer;
               callcode     : integer;
               priority, nesting, basepriority : integer;
               slice : integer ; time : integer;
               overtime : boolean;

         end;

type processref =@ PCB;
type processqueue = sequence of processref;
var ready = class
    var readyqueue : array [.0..maxpriority.]  of processqueue;
    procedure entry put (p:processref);
        var i:integer;
        begin i:=p @.priority;
              readyqueue [.i.] .put (p);
              interruptbus .set(i);
              interruptbus .enable(i);

        end;
    function entry get (i:integer): processref;
        begin select: = readyqueue [.i.] .get;
              if readyqueue [.i.] .empty then
              begin interruptbus. reset (i);
                    interruptbus. enable (i);
              end;
        end;
    begin var i: integer;
       for i: = 0 to maxpriority do
        begin readyqueue [.i.] .initialize;
              interruptbus . reset  (i);
              interruptbus . enable (i);

        end;
    end;
```

The procedure "enter" introduces a new process in the ready processes queue at the appropriate priority level and sets a signal with that priority on the interrupt bus. The function "select" gives as a value the pointer to the first ready process with the priority indicated as a parameter and resets the interrupt signal if the queue remains empty.

The two routines, together with the queue of ready processes, are organized as a class, "ready." At system loading, the data structure and the interrupt bus must be initialized as indicated by the initial operation.

In order to describe the routines it has been necessary to define two constants and three data types. The constants are the number of processor registers "maxreg," assumed to be 16, (this number is implementation-dependent) and the number of priority levels, "maxpriority." Priority 0 is considered the lowest and 15, the highest level.

A "processref" has been defined as the pointer to a process control block. "Processqueue" designates a queue of PCBs.

The PCB has been defined as a record providing items already discussed (succ, pred, priority, nesting, basepriority, registerarea) and several new items serving the following purposes:

- "callcode" records in a codified form the nature of a call to the kernel and serves to select the appropriate routine.
- "callparam" is an area used to transmit, if necessary, parameters to a kernel routine.
- "slice" and "time" indicate, respectively, the interval since the last process activation and the total time used by the process for the current program.
- "overtime" is a Boolean variable which becomes "true" when the process has consumed more CPU time than an allowed quantum, without being blocked.

The last three items of the PCB are not directly related to the kernel operations. They will serve for process scheduling in a time-sharing system and for accounting purposes.

When a processor enters or leaves a monitor or when it is preempted and allocated to another process, the processor uses a few routines in order to update the contents of the corresponding PCB(s). These routines are collected as a class, "runaid," and will be called during the execution of the monitor primitives.

In writing the routines, it is assumed that a processor has a register, "user," which contains the pointer to the PCB of the currently running process, and a register, "priority," which indicates the current priority of that process. Another register, "processorid," maintains permanently the processor identifier.

The coding of these routines follows.

```
var runaid = class;
    procedure entry incrnest;
       begin
          with user @ do
             begin nesting:= nesting + 1;
                Busmaster.request (processorid, 15);
             end;

       end;
    procedure entry decrnest;
       begin with user@ do
          begin  nesting := nesting-1;
             if nesting = 0 then
             begin priority := basepriority;
                busmaster.request (processorid, priority);
             end;
          end;
       end;
    procedure preempt;
        begin with user @ do
           begin registerarea := reg;
                priority := priorityreg;
                ready.put  (user);

           end;
        end;
```

```
procedure entry serve (i : integer);
    var p : processref;
    begin if user<>0 then preempt;
      p := ready.get (i);
    reg := p@. registerarea;
   user := p;
   busmaster.request (processorid, user @. priority);
end;
```

In a system where a time sharing operation is also possible, supplementary routines must be introduced for the processing of the clock interrupts. A request for the current priority bus is represented in the routines written above as a Concurrent Pascal-like statement:

busmaster. request (processorid, priority).

This statement specifies the important parameters: the processor identifier and the new priority level of the processor.

3.3.3. The implementation of the monitor promitives.

The preceding two subsections have provided the routines required for processor allocation and a few other auxiliary operations. It is now possible to write the routines that implement the monitor primitives described in Sec. 3.2.1. Since the kernel must use a separate semaphore for each monitor, there is an analogy between these routines and a Concurrent Pascal class. The name "gate" used for this class is borrowed from Ref.[1].

The coding is as follows:

```
type gate = class;
   var open : boolean;
      waiting : processqueue;
procedure entry enter;
    begin runaid.incrnest;
        if open then open: = false else
          begin waiting . put (user);
                user := 0;
          end;
    end;
```

```
procedure entry leave;
      var p: processref;
      begin if waiting . empty then
            open := true else
          begin p := waiting . get;
             ready . put (p);
          end;
          runaid.decrnest;

    end;
procedure entry delay;
    var p : processref;
    begin user@.reg:= reg;
       if not waiting . empty then
            begin p := waiting . get;
               ready . put (p);
            end else open := true;
       user := 0;
    end;
procedure entry continue (var p : processref);
      begin if p = nil then leave else
         begin ready . put (p);
              runaid . decrnest;
         end;
      end;
procedure entry inigate (var g :  gate);
      begin g:= new (gate);
            g@. initialize;
      end;
begin runaid . incrnest;
    open := false;
    waiting . initialize;
    runaid . decrnest;

end;
```

A supplementary routine, "initgate," is provided for use at system loading. It reserves the necessary memory space (this is the measing of the keyword "new" in "initgate") and then initializes a new gate.

3.3.4. A scenario for kernel calls.

While a process is executing on one of the system's CPUs, it will explicitly call a kernel routine if it needs to perform one of the following actions:

- obtain access to a monitor,
- leave a monitor,
- delay itself in a monitor queue,
- reactivate another process waiting in a monitor queue.

In addition to these situations, an executing process is abandoned and the kernel is called under two conditions:

- a process with a higher priority becomes ready,
- an interrupt signal with a higher priority arrives.

It will be assumed that the kernel routines are permanently kept in the main memory at fixed locations, so that the address at which a certain routine is found will always be known.

The actions performed by the requesting process in case of an explicit kernel call are now specified.

First, the address of the required kernel routine is stored in the PCB of the process as "callcode" while the necessary parameters (e.g., the identifier of the monitor to be entered) are introduced in the "callparam" area of the PCB. Subsequently, the processor issues a special instruction, TRAP, which causes the control to pass to a fixed address in the main memory. At that address an instruction, SRC (Sub routine Call), is provided with the address part pointing, in the indirect mode, to the "callcode" entry of the PCB. In this way the correct routine can be selected, and at the completion of that routine, the control returns to the address following SRC. There, a test should be performed to determine whether the processor has been preemted during the execution in the kernel (as after a "delay"). If this is true, a WAIT instruction must follow which causes processor to wait for the next interruption signal. Otherwise, a SRR (Sub routine Return) instruction passes the control back finally, to the point in the process where the kernel has been entered.

In Concurrent Pascal-like notation, these actions can be expressed as follows:

```
knexit: case user@ . callcode of
            1: enter;
            2: leave;
            3: delay;
            4: continue;
            5: initgate;
        end;
knexit: if user = 0 then
          begin busmaster . request (processorid, 0);
                wait;
          end;
        return;
```

An interrupt signal is processed in a similar way. This time, however, the processor receiving the interrupt must store the codification of the interrupt nature in an internal register. Subsequently the control passes to a fixed location in the main memory, other than the location used for explicit kernel calls. The processing of an I/O interrupt causes a new process to enter the ready queue. An interrupt for a ready process with a higher priority causes the processor to relinquish its current process, which is reintroduced in the queue of ready processes, and to start processing the new process. The return from the kernel takes place in the same way as for explicit kernel calls.

References

1. Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Tr. on S.E., Vol.1, No.2 (June 1975), pp.199-207.

2. Gonzales, M.J. and Ramamoorthy, C.V., "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," AFIPS Proceedings, FJCC 1966, pp. 1-15.

Chapter 4

THE FUNCTIONAL DESIGN OF A MULTIPROCESSOR MULTITASKING OPERATING SYSTEM

An operating system implemented in Concurrent Pascal appears as a program written in this language. The elements of a virtual machine capable of executing such a program have been explained in the previous chapter. From a very general point of view, the system hierarchy has in this case the following levels: the machine hardware, the kernel and the operating system written in Concurrent Pascal. Actually, the structure of the operating system is not directly influenced by the fact that the system has only one processor or is a multiprocessor: the kernel provides as many virtual processors as there are processes in the operating system.

Obviously, several functional levels can be identified in a more detailed examination of the operating system. The purpose of this chapter is to identify and describe these levels.

The approach of the general design of the system has been "bottom-up" (i.e., hardware - kernel - operating system). For the internal organization of the operating system, with the peculiar features of Concurrent Pascal in mind, a "top down" approach seems more suitable. It gives, in the author's opinion, a better understanding of the design requirements and ensures a clearer functional description, because it starts with the most general characteristics and gradually introduces new details.

4.1. General description of the operating system.

4.1.1 Objectives of the design

The most important feature of the system is its ability to accept requests from a user program for multiasking, i.e., for executing in parallel tasks of the same program. As already mentioned, the programmer or the compiler is responsible for dividing the program into tasks and for providing the information about the relationship between these tasks.

The operating system is able to accept requests for batch processing from a number of input units and to deliver the results of these requests to the indicated output units. The requests, as well as the results of their processing, are buffered on disk queues.

Another way to enter the system is from a set of time-sharing terminals. The time-sharing requests are either processed directly or, at the user's indication, introduced into the queue of batch requests. The directly processed time-sharing requests have precedence over the batch requests, in order to ensure an acceptable response time. It is assumed that these requests generally need a short processing time.

The system does not admit real time processing. The main purpose of this thesis is to develop an example system that is capable of processing parallel tasks, not necessarily a system with the most general structure.

Another limitation in the generality of the system is the fact that it accepts requests for parallel processing in the batch mode only. It is possible, however, to introduce such requests from a time-sharing terminal and route them to the batch input queue. This decision has been taken in order to simplify the overall structure of the system. Also, it has been estimated that parallel programs require a large amount of processing time, and therefore they would produce an unacceptably long response time if they were allowed to operate as time-sharing requests.

The parallel programs receive a higher priority in the batch input queue than the programs that indicate a strictly sequential processing. In this way, it is still possible to attain for the parallel programs a shorter turnaround time than would be the case if these programs were not divided into tasks.

Multiprogramming, in the sense of concurrent execution of several processes, is an intrinsic feature of an operating system written in Concurrent Pascal. The multitasking facility introduces a new dimension of multiprogramming, namely, between the tasks of the same program. The level of multiprogramming, i.e., the number of processes for which programs are simultaneously present in the main memory of the computer, must be determined by the memory capacity.

In a system operating without virtual memory, the degree of multiprogramming is limited by the fact that each process reserves the maximum amount of memory that it may need. In a system with virtual memory, however, too high a degree of multiprogramming may cause thrashing. Therefore, the system must be able to control the degree of multiprogramming in order to ensure that thrashing is avoided.

Each peripheral device of the system, the time-sharing terminals included, is under the control of a separate process. Algorithms have been developed for serving the requests to the sharable devices in such a way as to obtain a high degree of device utilization and a short average service time for individual requests.

The system implements a virtual memory in which a segment is the unit for memory sharing. Partially, information sharing is a result of the features of Concurrent Pascal. In addition, the system offers the possibility that several processes use, whenever feasible, the same copy in the main memory of a certain program. This is desirable, for instance, for an editor program accessed in common by the time-sharing users, or for a compiler called simultaneously by two or more processes.

Programs and data are stored on background memory units as files. A file may be either a segment or a collection of related segments. A file system catalogues the available information and controls the access to it. It must work in a very close cooperation with the virtual memory system. The file system uses the information supplied by processes in order to verify whether the current program has the right to access a particular file. The right of a program to access a file is established either by the programmer, if the file is created in that program, or is given by the system according to the category of the program.

4.1.2. The partition of the system activities into processes.

This section presents the types of processes that must exist in the system in order to provide the facilities stated as objectives of the design. It also specifies the monitors that are used for the direct communication between processes.

From the design objectives it results that the system operates basically in the batch processing mode. Therefore, the processes required for the batch mode are introduced first.

A spooling system introduces new jobs and displays the results of job processing. A job is a request from a user for a specific processing activity. It will generally imply a sequence of processing steps, e.g., the compilation of a source program, the preparation for execution of the obtained object program, and the actual execution of this program. The job is the unit of interaction between a user and the system. For a user accessing the system in time-sharing, a job is equivalent to a "terminal session," i.e., the actions performed from the moment that the user "logs in" until the user "logs out." However, a batch request originating from a terminal for batch processing becomes a separate, independent job.

The batch jobs are introduced from a number of card readers and are collected in an input queue on one of the disk units provided in the system. As Fig. 4.1. shows, each card reader is controlled by a card reader process which transmits card images to a buffer in the main memory. This buffer is implemented as a monitor and provides separate zones for several cards from each card reader process. If a buffer zone is full, the card reader process delays itself in a monitor queue.

An input process takes card images from the card reader buffer and transmits them a disk data buffer from which they will be transferred to the disk containing the input queue. The input process reactivates the delayed card reader processes. When the disk buffer is full, the input process enters a monitor with a request to obtain access to the disk.

It must be possible for the disk process to introduce data belonging to a job at the corresponding priority level in the batch input queue.

The system must ensure that each submitted job is eventually executed; thus,that there is a possibility to gradually increase the initial priority of a job. In this way, each job becomes the job with the highest priority in the input queue, and therefore it is selected for execution. The processes involved in the job execution are represented in Fig. 4.2.

A job scheduler process updates the priority of the jobs in the input queue and selects the job with the highest prioriy to be executed. The selected job is introduced in the ready jobs buffer, which is implemented as a monitor. A set of user processes (only two such processes are represented in Fig. 4.2) will take the ready jobs and control their execution. If there are no ready jobs, the user processes delay themselves in internal queues of the buffer. The operation of placing a new item in the ready jobs buffer always includes an attempt to activate a "sleeping" user process.

If a user process discovers during the execution of a job the existence of parallel tasks, the process will request the assistance of a coordinator. There are several coordinators in the system (only two are shown in Fig.4.2), which wait in a coordinator request monitor when they are inactive. A coordinator examines the relations between the tasks of a parallel program and sends the simultaneously executable tasks in the buffer for ready jobs. Subsequently, the coordinator is delayed in a queue of this monitor. The user processes will select for execution the parallel tasks and will report the completion of each task. After a task completion, the coordinator is reactivated to determine the effect of this task on the other tasks of the program. Whenever new tasks may start executing, the coordinator places them in the ready jobs buffer.

When the completion of the last task of a parallel program is reported, the coordinator that has been involved in the execution of that program delays itself in an internal queue of the coordinator request monitor.

The user processes and the coordinator are also provided with access rights to the disk containing the batch input queue, but this fact is not represented in Fig. 4.2.

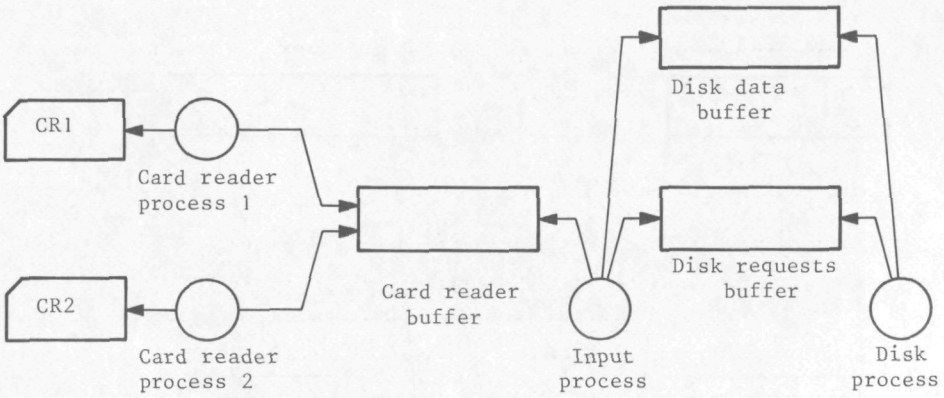


Fig. 4.1. Processes for building the batch input queue.

The access to the disk is controlled by a disk process. This examines the disk requests buffer where the pending requests are recorded and selects one of them according to a scheduling algorithm. Each request must specify the identity of the process that has issued it, the data buffer to be used and the disk address involved in the transfer. When a transfer is completed, the process that has initiated it is reactivated. If there are no other requests, the disk process is delayed in a queue of the disk requests buffer. From there it is reactivated when a new request arrives.

Note that two buffers are required for the communication between the input process and the disk process: one buffer containing the data to be transmitted and the other containing the requests for disk access. The same disk can be requested by several processes. Each process must then provide a separate data buffer, but there is only one buffer for access requests. It should also be noted in Fig. 4.1. and in the following figures of this section that arrows are used to indicate the access rights of a process, not the direction of information flow.

All the card reader processes have the same structure, because they perform identical functions. This means that all these processes can use the same copy of the programs directing the operation. The card reader processes are differentiated by assigning an index to each process.

The batch input queue is organized into several subqueues, as determined by the priority allocated to each job. The priority of a job is calculated by the input process, based on several parameters that the user must provide on a job control card which is always the first card of a new job.

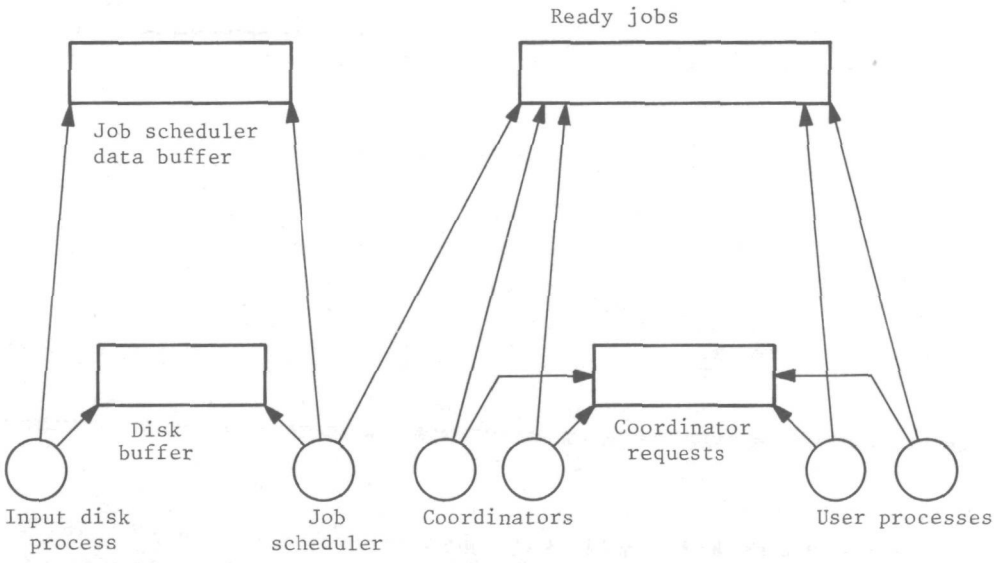


Fig. 4.2. Processes involved in job scheduling.

The number of coordinators and user processes in a particular system are established during the system initialization, by taking into account the physical resources of the system. The buffer of ready jobs is organized into several priority levels. Tasks originating from parallel programs will have a higher priority in this buffer than the ordinary jobs.

Figure 4.3. shows the relationship between the user processes and the processes of the disks containing the input and the output queues, respectively. For simplicity it is assumed that the two queues are constructed on separate disk units, but it is possible to design a system having both queues on the same disk.

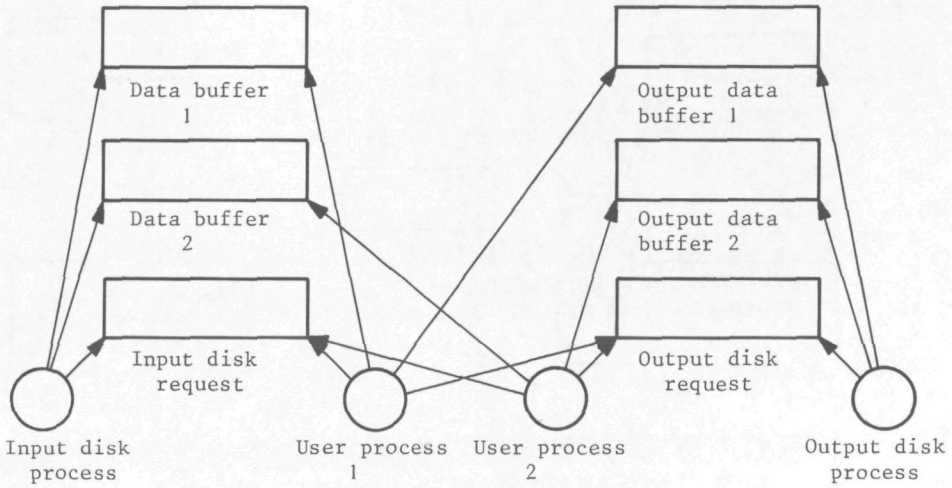


Fig. 4.3. Access rights of user processes.

The output queue contains the results of job processing that must be printed or otherwise transmitted to the users.

With the assumption that only printers are available, the processes involved in printing the results are represented in Figure 4.4.

An output process is responsible for taking out entries from the output queue and for sending them to a set of printer processes. Notice that a job may request that its results be sent to a particular printer and, therefore, that the output process must be capable of recognizing such a request. A line printer process is, in a way, similar to a card reader process. It takes the image of a line from the printer buffer and controls the actual printing of the line. When the corresponding slot in the buffer is empty, the printer process is delayed until the next request arrives.

Apart from the two disk units dedicated to the spooling system, the computer has other disk units, as well as magnetic tapes and other types of peripheral devices. It is assumed that the tape units are used mainly as intermediary devices for creating or reading new files or for off-line storage. The collection of files directly accessible in the system is found on the disk units.

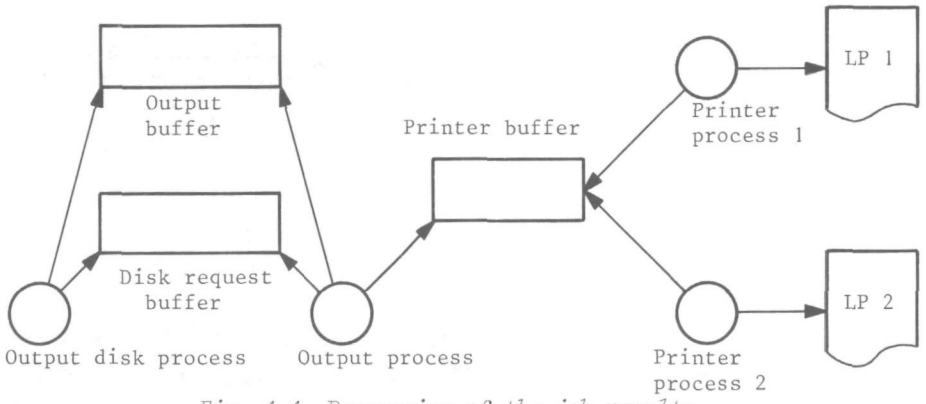


Fig. 4.4. Processing of the job results.

A file catalog, implemented as a monitor, maintains a hierarchy of file directories to help in accessing the desired file. There will be a separate disk process for each disk of the system, and the role of the file catalog is to activate the process for the disk containing the file requested by a user. A more detailed description is given later in the section dedicated to the virtual memory.

The access rights of a terminal user process are represented graphically in Fig. 4.5.

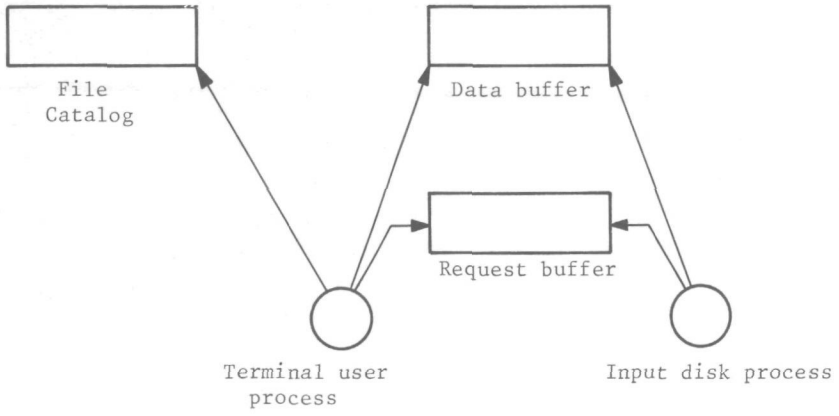


Fig. 4.5. The access rights of a terminal user process.

Each terminal is represented by a process denoted as a terminal user. A terminal user process must have the possibility of communicating with the disk holding the input queue in order to route jobs for batch processing. Further, the terminal user process must have access to the file catalog and must be able to call several standard programs (e.g., the compilers, the editor). It must be able to interpret a set of commands introduced by the user which are in principle identical to the job control cards.

4.2. Virtual memory organization.

The address space of a program, i.e., the information that the program may access, is partially established during compilation. When the program is executed, the interface routines, provided by the process that controls the execution, enable the program to access other units of information in addition to its own routines and data. Specifically, one of the interface routines allows the program to call other programs, if it can supply a valid program identifier and the required parameters. In this way, the address space varies during execution.

Compiled programs are stored as files in the file system of the computer. A compiler cannot verify the legitimacy of particular program calls because it may happen, for instance, that the called program does not yet exist in the system at that moment. The compiler can only verify that the call is made according to the rules imposed by the interface procedure. At run time, the called program must verify at least that the parameters supplied by the caller correspond in number and type with the formal parameters in the definition of the program.

Because every program is executed under the control of a certain process, it is convenient to consider the address space to be a property of the process. In principle, the address space of a process may contain the entire information available in a computer system.

Each process can regard the data defining its address space to be private information and, therefore, defined as a class. This class will be called the virtual memory of the process. It implements the mapping of information between the file system and the main memory of the computer. For this purpose, the virtual memory class has access to two monitors: file catalog and core monitor.

The file catalog is able to find the location of the file required by a program and to supply certain information about the file to the virtual memory class.

The core monitor maintains an account of the allocation of the main memory space and has the ability to allocate space for new requests and to reclaim unused space. It also initiates the I/O operations required for the transfer of information between the main memory and the secondary memory.

4.2.1. The class "virtual memory."

The internal organization of the virtual memory class is influenced by the information addressing technique used in the computer and by the provisions concerning the information protection and sharing. It is assumed henceforth that the information is divided into segments of variable size and that a segment is the unit for information sharing. A single main memory copy of a segment is used by all the processes that must access that segment.

The system discussed in this thesis uses an addressing technique which is based on capabilities. This technique is in many ways similar to that provided in the Plessey System 250. A capability is essentially a unique identifier of a segment, for instance the value of the system clock at the moment when the segment was created. During the entire existence of a certain system, the same identifier cannot be attributed to two or more segments.

The segments are further divided into fix-sized pages. Since a segment in this system contains generally a program routine or a small number of data items or capabilities, the page size should not be larger than 128 or (in the extreme) 256 computer words.

A program and its private data are organized as a package consisting of several segments. A segment contains either capabilities or data and program code. A program package should always have a Primary Capability Segment (PCS) which contains capabilities for all the segments directly accessible to that package. These segments contain the code of the program routines, the program data or capabilities for other segments.

A Process Central Capability Table (PCCT) is associated with a process. This PCCT consists of entries defining the monitors, classes and the interface routines that the process may access, as well as the PCS of the cur-

rently active program. When another program is called, the capability for the PCS of the old program must be saved for a subsequent return. For this purpose, the process has a program stack, pointed to by another entry of the PCCT.

A stack frame is defined whenever a program is called. The first information introduced in a new stack should be the capability for the PCS of the old program. Further, the space in the new stack frame is used in calling the program routines and for storing the local variables of these routines. The processors must have a "stack-relative" addressing mode for the access to local variables.

Further details about the addressing are given with the aid of Fig. 4.6.

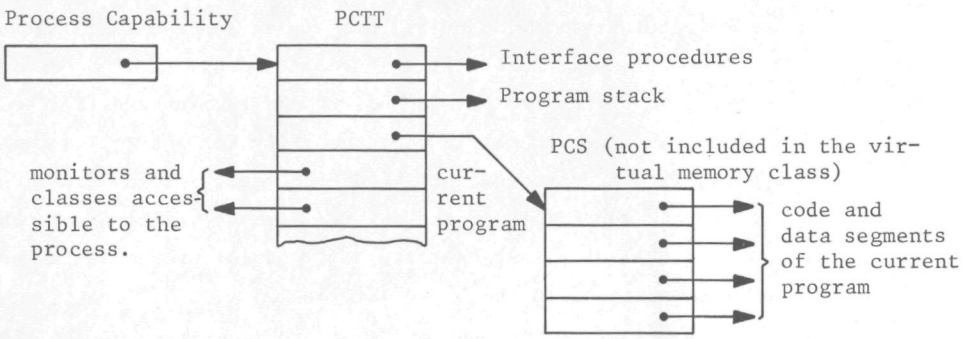


Fig. 4.6. Addressing structure in the proposed system.

Each entry in the PCCT has a predetermined function. The first entry always points to the PCS defining the interface procedures. The second contains the capability for the program stack, and the third entry points to the PCS of the current program. Further, there are entries pointing to the monitors and classes accessible to this process. The allocation of the entries in the last category is process-dependent and will be performed by the Concurrent Pascal compiler during the compilation of the process definition. Because the interface routines are also included in the process definition, it is possible to completely specify the virtual addresses required in the interface routines. The compilers for all other programming languages possibly

used in the system should know only the conventions for the first three entries in the PCCT.

To the items of a Process Control Block, described in the previous chapter, should be added a capability for the PCCT. When a process is executing, this capability is stored in an internal processor register, indicated as Process Capability in Fig. 4.6.

A virtual address in this system consists of the following fields:

- "Package," used as an index in the PCCT for selecting the desired program package;
- "Segment," the index of the desired segment in the PCS of its package;
- "Page," the index of the desired page in the segment;
- "Displacement," indicating a particular memory word in the page.

The transformation of a virtual address into a real address is illustrated in Fig. 4.7. The first step of the transformation consists in accessing the PCCT (as determined by the Process Capability and the Package field of the virtual address) in order to obtain the capability for the appropriate PCS.

Using this capability and the contents of the "Segment" field, a CPU can obtain from the PCS the capability of the desired segment. With this capability, the Page Table (PT) of the segment can be accessed, producing the base address of the page indicated in the "Page" field.

The "Displacement" field is used to select, in the last step of the transformation, the particular memory word.

In order to reduce the time consumed in the address transformation, each processor should be equipped with a set of high-speed registers which store the capabilities of the most recently used segments and pages.

A capability for a certain segment will indicate whether the segment is active or not. A segment is said to be active when it has a PT in the main memory. If the segment is not yet active, a routine of the virtual memory class obtains the control and calls the core monitor to activate the segment and bring (part of) it into the main memory. While the transfer is in progress, the requesting process must be blocked. When a process calls a new program, it must receive a capability for the PCS of that program. The name of this program, supplied as a parameter by the calles, is transmitted by the virtual memory to the file system. This searches the program name in the file directories and returns a capability for the PCS of the program and indicates whether this segment is already active, due to

the program being already used by another process. At this stage, it is also possible to verify whether a certain user (whose identity is provided by the process) has the right to access the program.

If the PCS of the new program is already active, the process can continue. Otherwise, the segment must be activated by the core monitor.

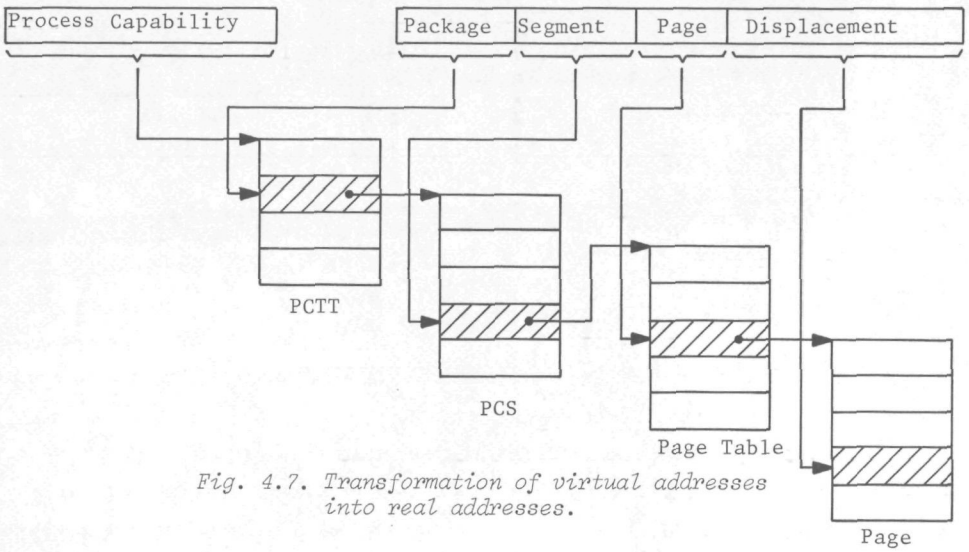


Fig. 4.7. Transformation of virtual addresses into real addresses.

4.1.2. The core monitor.

The core monitor administers the main memory space and initiates the I/O operations for the information transfer between the main memory and the secondary memory.

The core monitor operates mainly on the information maintained in two tables. The first table, called the System Capability Table (SCT), contains the capabilities and the Page Tables of the active segments. The second table, called the System Page Table (SPT) provides an entry for each page frame of the main memory.

The organization of these tables is explained with the aid of Fig. 4.8.

The SCT is a "hash" table, accessed by using the capability of a segment as the hash key. An entry in this table, denoted as SCTE, contains the following information about a segment: the capability, the address in the secondary memory, the length, the access mode, the Page Table and a reference counter.

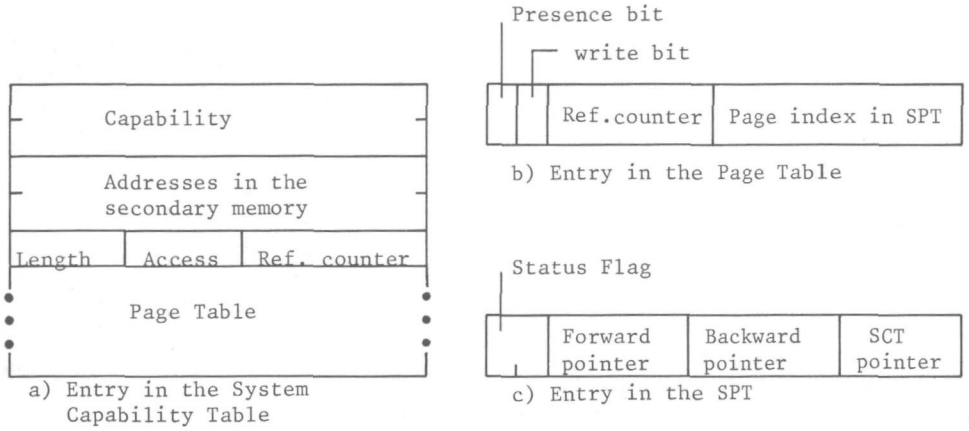


Fig. 4.8. Some data structures used in the core monitor.

A segment has a fixed address in the secondary memory. For an active segment, this address is stored in the SCTE and can be used when a page of the segment must be transferred between the main memory and the secondary memory.

The reference counter indicates the number of processors which currently have copies of the segment capability in one of their internal registers. This information is used in the allocation and de-allocation of entries in the SCT and of page frames in the main memory. A segment whose reference counter has the value zero is said to be an unreferenced segment.

Because the entries in the SCT have a fixed length, the system imposes a maximum size for the segments: a segment cannot have more pages than the number of entries in the Page Table. The optimum values for the number of entries in the Page Table and for the size of a page should be determined from studying the average segment length and the reference pattern distribution in the system. Such a study has not been performed in this thesis. It may be assumed, however, that with a page size of 128 words, the page table need not provide more than 32 entries.

There is also a reference counter for each page, having the same meaning as the reference counter of a segment. However, it will be used only for the allocation of the page frames in the main memory. A page whose reference

counter has the value zero is an unreferenced page.

Further, an entry in the Page Table of a segment contains a presence bit and a write bit. The presence bit contains a "1" when the page is in the main memory and a "0" when it is not. A value of "1" for the write bit indicates that the page has been modified since the last time that it was brought into the main memory. The last item in the Page Table is the index of the page frame of the main memory allocated to this page. It is used as an index for accessing the SPT and, at the same time, as the base address of the page for the translation of virtual addresses into real addresses.

The SCTE contains two control fields, defining the length of the segment and the type of access permitted to the segment. Both fields are used during the address translation.

The second table of the core monitor is the System Page Table (SPT). An entry in this table contains the following information about the corresponding page frame: a status flag, two pointers for linking in a list and a pointer to the SCTE of the segment to which the page belongs.

The status flag consists of two bits which indicate the page status according to the code shown below:

00 - free page;
01 - page being transferred;
10 - unreferenced page;
11 - page in use.

The pointers are used to connect into one list the free pages and, into a separate list, those pages which are still present in the main memory, but not currently referenced by any processor (unreferenced pages). The core monitor should keep, as separate variables, pointers to the head and the tail of each of the two lists. A new entry is always appended at the tail of a list, and the entry selected for use when requested is always taken from the head of the list. Therefore, this is the entry that has stayed for the longest time in the list.

The pointer to the SCTE is used during the removal of a page from the main memory.

It can be stated that the core monitor is basically concerned with the translation of virtual addresses into real addresses. In this operation, the core monitor is aided by the hardware of the system processors. In the solution developed here, it is assumed that the only hardware aid for the core

monitor is a set of high-speed registers denoted henceforth as Address Translation Registers (ATS), with which all CPUs are equipped.

Further support means could be incorporated in the processors-memory interconnection scheme, if required.

The structure and the use of ATRs are illustrated in Fig. 4.9. An ATR consists of an associative section and a conventional (random access) section. The associative section has fields for the package, segment and page indices. The random access section contains the segment capability, the page frame index and a reference counter.

The ATRs are used for indicating the most recently referenced pages during the execution of a certain process. When a virtual address is developed in a CPU, the package, segment and page fields of the address are compared with the corresponding fields in the associative section of the ATRs. If a match occurs in an ATR, the page frame index of that register is concatenated with the displacement value contained in the virtual address, producing immediately the real address. The core monitor need not be accessed, saving thereby execution time.

If the comparison does not produce a match, the virtual memory class and/or the core monitor must be called for help. At the same time, an ATR should be reserved for the newly referenced page. The reference counter, which consists of a few binary positions (values between 4 and 8 are considered at the most probable) indicates which register will be selected.

Several criteria can be used for the selection. One of them is based on incrementing the counter by 1 for every access to the page represented in that ATR. Whenever the counter passes through zero, the counters of all ATRs are reset to zero. The counters are also reset to zero for each new page descriptor introduced in one of the registers.

In this way it is possible to keep an account of the reference frequency to all the pages currently in use in the processor. The least frequently used page (i.e., the page described in the register with the lowest value in the reference counter) is selected for replacement. When more than one register counter contains the minimum value, the processor selects one of these registers at random.

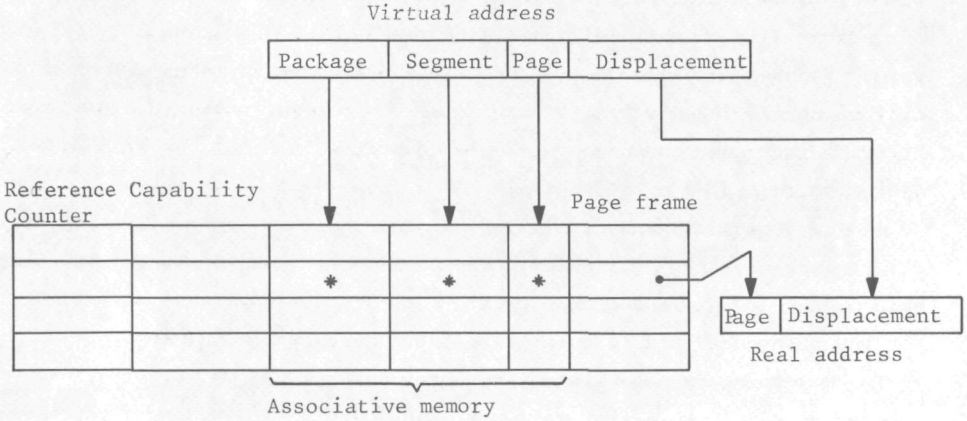


Fig. 4.9. Address translation with the aid of the ATRs.

If no ATR contains a capability for the requested segment (such a situation is a "segment fault"), the virtual memory class of the process should be called to produce this capability. Otherwise, the capability is obtained from an ATR, which means that only a "page fault" has occurred. In calling the core monitor, the segment capability and the page index are provided as parameters. The core monitor must determine whether the addressed page is present in the main memory. Therefore, it uses the segment capability to access the SCT. The result can be that there is already an entry in this table for the requested segment of that the segment is not active.

Assume that the segment is active. In this case, with the page index transmitted as a parameter, the core monitor accesses the Page Table of the segment. The presence bit in the Page Table entry is first examined. If this bit is zero, the page is not in the main memory and a page transfer should be initiated. Otherwise, the operation continues by examining the reference counter.

A strictly positive value of the reference counter indicates that the page is certainly in the main memory. Therefore, the page frame index stored in this entry can be transmitted to the processor. Simultaneously, the reference counter of the page is increased by one. Note that when the segment capability is obtained from the virtual memory class, the reference counter of the segment should also be increased.

If the reference counter of the page is zero, the core monitor accesses

the SPT to determine whether the page is still in the main memory, This is indicated by the bits of the status flag, which should be 10. If so, the value is changed to 11 (page in use), the SPT entry is taken out of the list of unreferenced pages, and the core monitor enters again the SCT. Here, it increases the page reference counter completing, by this, the operation requested by a process.

If the status flag has the value 01 when the core monitor examines the SPT, the page is not available in the main memory. The presence bit in the SCTE must be reset, and then the core monitor must reserve a page frame for the referenced page and initiate the necessary I/O transfer.

In reserving a page frame, the core monitor examines first the list of free pages. If this list is empty, the monitor selects the first entry. Otherwise, it selects a page from the list of unreferenced pages. The value of the status flag in the selected entry becomes 01. The core monitor must also examine the write bit of this page and, if the bit is "1," transfer the page back into the secondary memory (at the address indicated in the SCT). Finally, the core monitor initiates the transfer of the new page.

A process waits in a core monitor queue until the I/O transfer, that it has requested, is completed. In this way, other processes are allowed to enter the core monitor.

Until now it has been assumed that the referenced segment is active. If this is not the case, the core monitor should first activate the segment. With the segment capability supplied by the calling process as a parameter, the core monitor tries to find a free entry in the corresponding hash zone. If the core monitor cannot find a free entry, it searches for a segment, in the same zone, with a zero reference counter. Such a segment, if found, can still have some pages in the main memory (in the unreferenced state). Therefore, its Page Table is scanned. When the presence bit of a page is 1, the core monitor resets this bit and then accesses the SPT to set the status flag to 00 and to introduce the entry in the list of free pages.

In the case that no entry can be found for the new segment, the requesting process is delayed and the request is introduced in a list where it waits until the reference counter of a segment in the corresponding hash zone become zero.

Now the situations in which the reference counters should be decremented will be examined.

Obviously, the reference counters are decremented when an ATR is de-allocated. It is relatively easy to determine by hardware means whether this has been the last register addressing a certain segment. If so, both the segment reference counter and the page reference counter are decremented. Otherwise, only the page reference counter is affected.

The reference counters are also decremented when a process is delayed. Therefore, each "delay" operation includes a call to the core monitor, if the process is not currently executing in the core monitor. The time that a process remains delayed is always assumed to be long in comparison with the time required for process switching. The core monitor examines, for a "delay" operation, all the ATRs and reduces the corresponding reference counters. In this way, the memory occupied by a delayed process can be re-used for other processes.

For the duration that a process waits in a semaphore queue before obtaining access to a monitor (short-term scheduling), the reference counters are not modified, although the processor is, possibly, re-allocated to another process. A general assumption has been that all monitor routines are short and, therefore, the waiting time in a semaphore queue is not long enough to justify the de-activation of pages and segments. Instead, the contents of the ATRs are saved on the process stack whenever the processor is re-allocated. The first operation of a new process should be the reloading of the ATRs.

The core monitor is also called when an I/O process completes the transfer of a page. It changes the status flag of the page (the page index is given as a parameter by the I/O process) to 11 and continues the process that has requested the page. The presence bit in the Page Table of the SCTE must be set to 1.

Until now it has been tacitly assumed that only one process waits for a certain page to be transmitted into the main memory or for a certain segment to be activated. In the following a description is given of how the core monitor treats the cases when more than one process is waiting for the same segment or page.

Processes waiting for the activation of the same segment are linked into a list. When the request can be satisfied, the core monitor continues the first process in the list. This process receives, however, a parameter specifying that there are more processes waiting, and therefore it calls again

the core monitor for the continuation of another process. The procedure is repeated until the list is empty.

The processes waiting for a page are treated in a similar way when the transfer of that page is completed.

With the increase in the number of processors and in the size of the main memory in a system, the core monitor can become a bottleneck in the system operation. A possible solution for avoiding this is suggested below. This solution divides the functions of the core monitor between several separate monitors, organized as shown in Fig. 4.10.

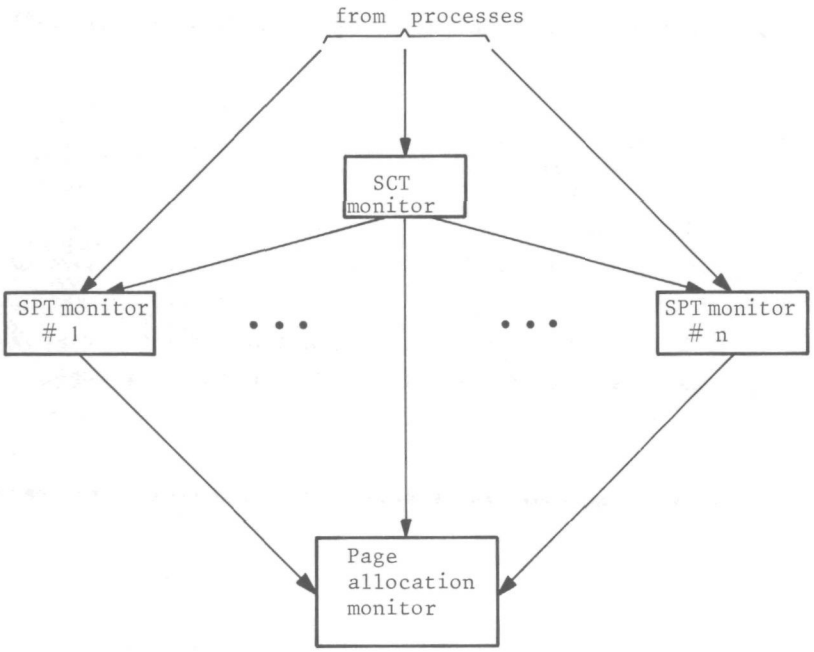


Fig. 4.10. An alternative organization for memory management.

The page reference counters are now maintained in the System Page Table, which is divided between n SPT monitors, one monitor for each memory block.

A process calls the SCT monitor whenever a new page descriptor must be introduced in an ATR. But the process operates in this monitor only for a short time, to obtain the index of the page frame, if the page is present in the main memory. Subsequently, the SPT monitor takes care of the updating of the

reference counters or, when the page is not present in the main memory, of the allocation of a page frame with the aid of the page allocation monitor. The page allocation monitor operates on the lists of free and unreferenced pages.

During the execution in the core monitor, no page faults or segment faults may occur. A straightforward technique which ensures this consists of providing the processors with the possibility of specifying whether the address calculated during an instruction is virtual or real.

For real addresses, the translation mechanism is not used. All the operations in the core monitor must specify real addresses. Therefore, the routines and data structures of the core monitor must be permanently kept in the main memory. The kernel routines and data structures should also be permanently present in the main memory.

Chapter 5

A SIMULATION MODEL OF THE PROPOSED COMPUTER SYSTEM

This chapter begins with some general remarks about the modelling and simulation of computer systems. Subsequently, it contains a simulation model of the system proposed in the previous two chapters. Specific problems encountered during the construction of the model are emphasized. The results obtained from a simulation program written in SIMULA 67 are presented and discussed in the last section of the chapter.

5.1. Computer systems modelling and simulation.

The evaluation of the performance of an existing computer system is based on measuring the adequate system parameters: the time that the CPU is active, the amount of main memory occupied at every moment, the utilization factor of each I/O channel, etc. From the measurement results, variables which indicate the quality of system performance to the designer or user can be obtained, e.g., equipment utilization, program service time, etc. However, a study into the possibilities of modifying or extending an existing system or a study of a new design must be based on a system model. The model reproduces the features of the proposed system, at the level of detail desired by the model designer. The model will predict the values of the essential performance variables and the influence that modification of certain system parameters (CPU speed, memory capacity, number of I/O channels, scheduling algorithms, etc.) will have on the values of the

performance variables.

There are two basic types of computer system models:

- Analytic models, developed as extensions of the queuing theory, which aim at determining some mathematical relations between the performance variables and the system parameters.
- Simulation models, which aim at reproducing the dynamic behavior of the system.

Studying a system with simulation models has an experimental nature: the events of interest are recorded during the experiments and the system performance variables are obtained by the statistical analysis of the recorded events.

In both types of models, the necessary calculations are almost always performed with the aid of a computer. Therefore, the models will be translated generally into computer programs.

The analytic models describe the system behavior in a simplified form. Nevertheless, a well-designed model can give very useful information which is often sufficient to evaluate system performance. Moreover, the analytic models need less computing time than the equivalent simulation models.

When a more detailed reproduction of the system behavior is required or when the mathematical relations between the system parameters and the performance variables cannot be easily expressed, a simulation model should be used. In this case, the simulation experiments must be planned judiciously. If wise planning does not occur, too much computing time will be required and the danger exists that the experiments might not produce the necessary information.

Every computer model includes a description of the system workload, in addition to the description of the system structure and operation. The workload is the collection of user jobs that are submitted for execution in a certain period of time. It is important that the workload in a computer model has the same characteristics as the real workload. Only in this case, assuming that the system structure and operation are correctly represented, can the modelling results be valid.

In constructing a simulation model it is also useful to have an adequate method for describing the system structure and the relations that exist between the various components during operation. This is particularly important for the models of complex, general-purpose operating systems. A basic requirement of the description method is its property that it allows a

straightforward translation of the model into a computer program.

Various programming languages are used in modelling. The analytic models are solved in most cases with the aid of a general-purpose programming language.However, special languages have been developed for simulation, but general-purpose languages are also used. The aim of the special-purpose languages is to simplify the writing of the simulation program and, to a lesser extent, to reduce execution time.
The model described in the next section has been developed based on these general considerations.

5.2. A description of the developed simulation model.

One of the features of the computer system proposed in the preceding chapters is multitasking. This feature has been introduced with the purpose of reducing the turnaround time of some large programs by dividing them into simultaneously executable tasks, i.e., by writing them as parallel programs. In a multiprocessor system, the simultaneously executable tasks may indeed be processed in parallel.

Obviously, the turnaround time of the parallel programs will be reduced in comparison with the turnaround time of the same programs in a system without multitasking. It is of interest, however, to study how significant this reduction is and how multitasking influences the service that other types of programs receive from the system. Not all parallel programs will have the same internal structure, i.e., the same number of tasks and relations between tasks.Therefore,various structures for parallel programs should also be taken into account in the system study.

A simulation model is preferable to an analytic model in this case.

Especially the different structures of the parallel programs are difficult to represent in an analytic model, but there are also other features, like the scheduling algorithms, which suggest the use of a simulation model.

By studying some of the available methods for elaborating a simulation model, the author has come to the conclusion that the "evaluation nets" (E-nets) described in Ref. [3] (see also Appendix 2) are the most convenient for representing the simultaneous actions that take place in an operating system.

The developed model does not reproduce in detail the structure of the operating system designed in Chapter 4. It only describes the flow of the user programs through the system. This simplification has been considered permissible due to the restricted purpose of the model: to give indications about the effect of multitasking on the service received by the user programs. Moreover, the model is constructed in such a way that the system overhead remains approximately the same, regardless of whether the system uses multitasking or not. Therefore, the system overhead is left out of consideration in this first version of the model.

Other simplifications introduced in the model concern the main memory and the internal structure of the parallel programs.

Memory constraints are not taken into account; the main memory capacity is considered to be sufficient for storing the information required by all active processes. The effect of information sharing between processes is discarded based on the same consideration given for the system overhead.

In specifying the system workload, only a small number of different parallel program structures have been considered. The model requires that each parallel program structure be expressed as a separate E-net (a macro E-net in the terminology introduced in Appendix 2). For practical reasons, only a limited set of such macro E-nets could be developed. The selected parallel structures contain, however, the types of inter-task relations supposed to appear most often in real programs.

The configuration of the modelled system is represented schematically in Fig. 5.1. The input devices for the batch jobs are card readers, and the job results are sent to a number of line printers. The time sharing jobs enter the system from remote terminals. The secondary storage consists of disk devices only.

The number of terminals, card readers, line printers, disks and CPUs are parameters of the model.

A simplified version of the model is illustrated in Fig. 5.2, which is explained below.

First the symbols used in Fig. 5.2 will be explained briefly. The E-nets operate with locations and transitions. A location, depicted graphically by a circle, represents a certain condition in the system, which can be satisfied or not at a given moment. A transition is depicted by a vertical line and represents an activity of the system. A transition has input locations

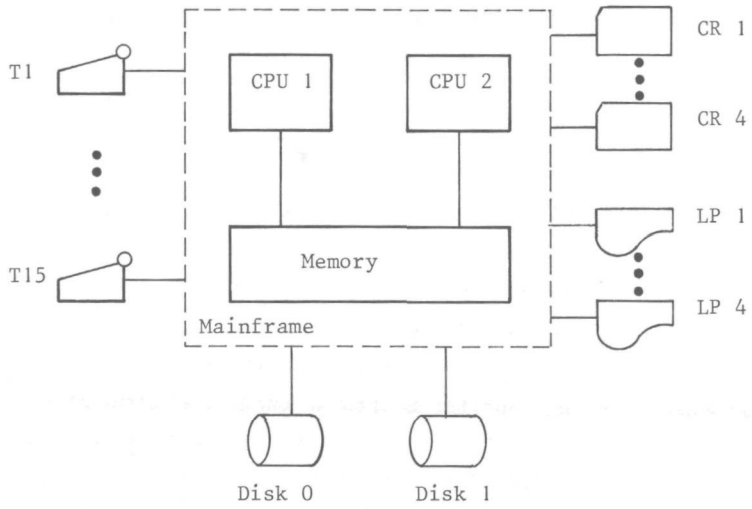


Fig. 5.1. System configuration for simulation experiments.

and output locations. When the conditions represented by the input locations are satisfied, the transition "fires" and modifies the state of its output locations. One can say, therefore, that the locations describe the system state; whereas the transitions describe the dynamic behavior of the system from one state to another. Rectangles in Fig. 5.2. denote macro E-nets, i.e., parts of the system that have not been represented at the level of detail required by the locations and transitions. The rectangles denoted A1 and A2 are token absorbers and those whose name starts with a "G" are token generators. A token is a dot which, when present in a location, signifies that the condition represented by that location is satisfied. A special symbol, an ovoid, is used to represents queues (macro-locations). Pairs of numbered triangles are used as connectors between the various parts of the model.

The input and output locations are connected to a transition by arrows. A small vertical bar on an arrow denotes a conditional path for the tokens.

In the model, the system terminals are represented as token generators, GT_1 - GT_n. They produce terminal requests (also called time-sharing requests) which are gathered into a "terminal requests queue," QTERM.

Bibliotheek TU Delft

Magazijnslip/*Picking slip*

Ik wil dit document graag verlengen/*I would like to renew this loan.*

in te vullen door de Bibliotheek TU Delft/*to be completed bij the Delft University of Technology Library*

☐ verlengd tot/*renewed until :*

☐ niet verlengbaar, a.u.b. retour/*renewal not possible, please return the document*

☐ opmerkingen/*other :*

Briefkaart

Aan de Bibliotheek van de
Technische Universiteit Del
Postbus 98
2600 MG DELFT

FRA

BRI

DE BLAEUWE WERELT APELDOORN

Geleend op/*Date of issue*:
15 15 15 10 1991

Terugbezorgd op/*Returned on*:

Uiterlijk terugbezorgen op/*Date due*:
12 11 1991

Paraaf/*Initials:*

Documentgegevens/*Received on loan*:

Jurca, I.
MULTIPROCESSOR SYSTEM WITH MULTITASKING FACILITIES

1200425 CBmg bkuitln 248254

Gebruikersgegevens/*Borrower* :

14618 Bruin, J.A.R. de

Seringenhove 17
2295 RD Kwintsheul

1117.624.5 nee

The batch input stations (card readers) are also token generators, GCR_1-GCR_m. The generated batch jobs are written into an "input queue," INPUT, maintained on the disk unit 0. The operation of the disk units is not represented in Fig. 5.2. This figure shows, however, that a write request is sent to the disk through connector 1 and that the disk reports the completion of the request through connector 2. A token placed in location b_2 signifies that a new job has been introduced in the input queue.

The system provides a number of process control blocks (PCBs) for the user processes. The free PCBs will be found in a queue, denoted in Fig. 5.2 as PCBQ.Once a PCB (thus a user process) has been allocated to a job or task, it remains allocated until the job or task is completely executed.

The jobs are divided into three priority levels: time-sharing requests (each request is considered as an independent job) have the highest priority, followed by parallel programs and then by the simple batch jobs.

The Parallel Programs Manager (PPM) provides several coordinators which have the role described earlier in Chapter 4. The number of simultaneously active parallel programs is limited by the number of coordinators. Because the batch input queue is processed in FIFO order, the input queue processing is temporarily stopped when a parallel program appears at the head of the input queue and no coordinator is available.

The PPM collects in the queue b_4 all parallel tasks ready to be executed. In order to decide which job will be activated when a PCB becomes available, the general job scheduler (transition a_6) examines the terminal requests queue, the parallel tasks queue and the location b_5 (indicating a "simple batch" request), in this order. It selects the request at the head of the first non-empty queue or, when both queues are empty, the simple batch request.

All active programs, i.e., those having a PCB allocated, enter the CPU queue (CPUQ) where they wait for a free CPU. When a CPU becomes free, it takes the first job from the CPUQ in FIFO order. If the queue is empty, the CPU remains idle. Once execution of a program has started in CPU, that program maintains control of the CPU until an I/O operation is requested or the program completes execution.

When an I/O operation is requested, the CPU directs the request to the corresponding peripheral device (here a disk) and then tries to start the next job in the CPU queue. Programs for which an I/O operation is completed, are returned to the tail of the CPU queue.

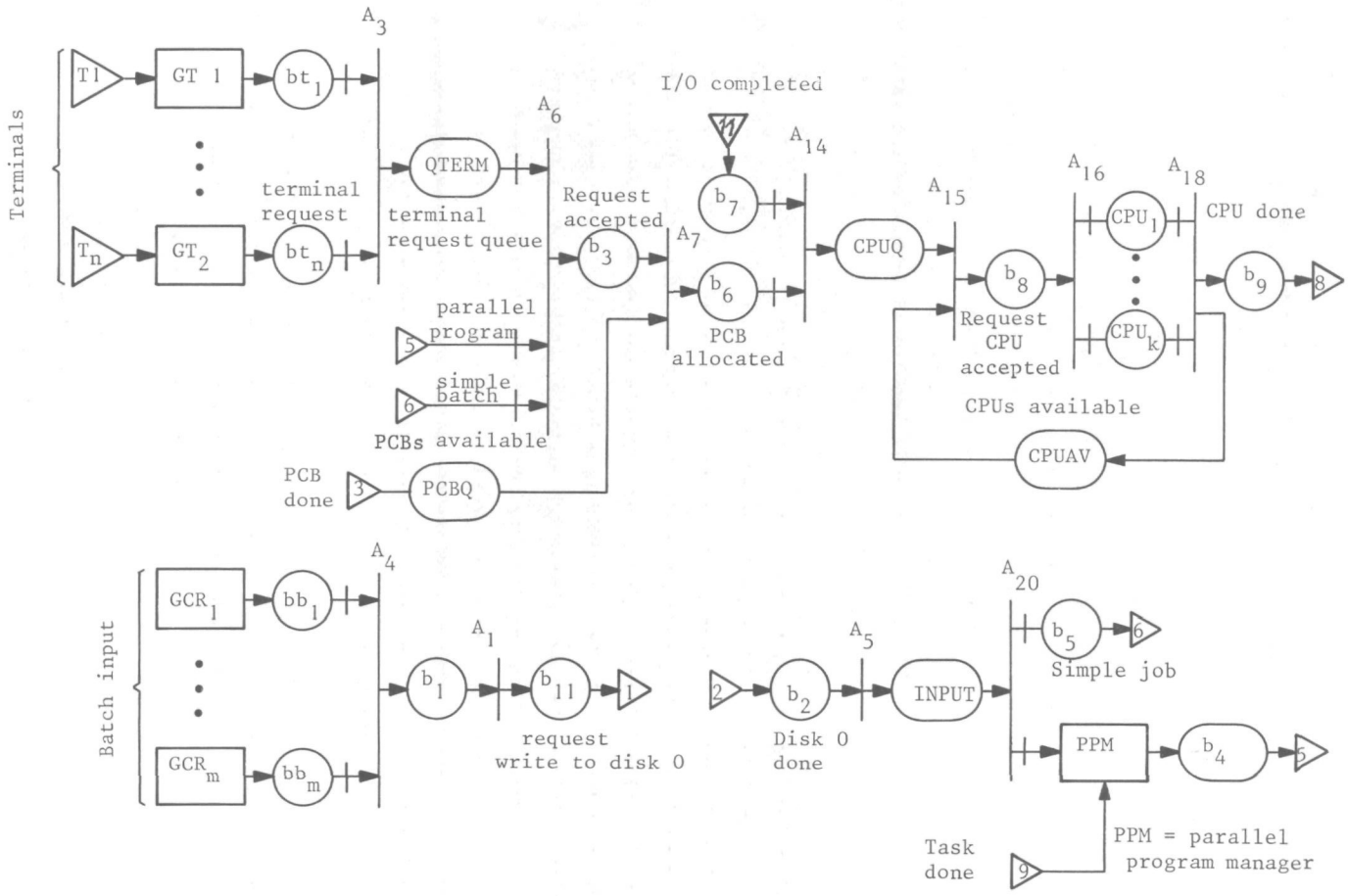


Fig. 5.2. Simulation model constructed with E-nets.

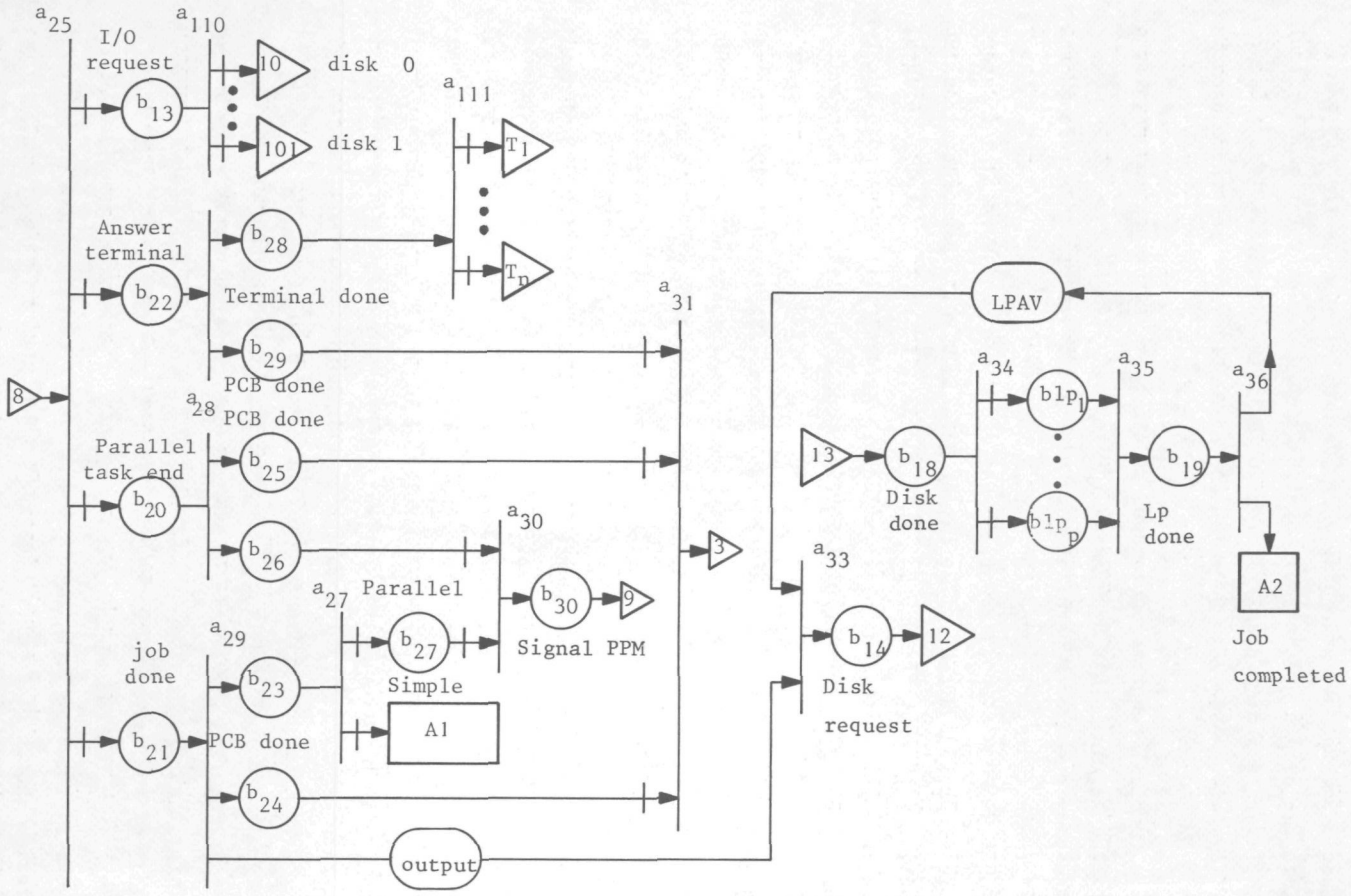


Fig. 5.2(contd.). Simulation model constructed with E-nets.

When a CPU is released because of job completion, the transition a_{25} examines the nature of the job: terminal request, parallel task or "simple" job(the last task of a parallel program is included in this last category). In all cases, the PCB must be released (transition a_{31}) and introduced in PCBQ.

For terminal requests, an answer should be sent through the transition a_{11} to the terminal that has issued the request.

The completion of a parallel task is signalled to the PPM through the transition a_{30} and the connector 9.

For job completion, a notice that a new set of job results if available is first introduced in the output queue, OUTPUT. Then, the transition a_{27} verifies whether the job has been the last task of a parallel program or a simple batch job. In the first case, a signal will be sent to the PPM. In the second case, nothing will be done and, therefore, the token will be absorbed in A1.

From the output queue, the job results are sent to an available line printer. The line printer issues disk requests to obtain the results and, when the printing is completed, introduces a new dot in the queue of available line printers, LPAV. A dot is also absorbed in A2, denoting that the job has left the system.

Special problems have occurred in the workload specification, because no real system has been available for measurements. For terminal requests and for simple batch jobs, some data indicated in the literature [4] have been used. However, no data have been found about parallel programs. Therefore, a number of "patterns" of parallel programs have been devised for this model. One such pattern (the simplest) is represented in Fig. 5.3.a, using again E-nets. Figure 5.3.b shows the relationship between the four tasks of the program. A short explanation of the pattern follows.

Signals are received from a coordinator through connector 30. The token placed in location BP3 specifies whether the signal concerns the start of the first task or the completion of one of the tasks. In the last case, the identity of the task is also given.

The first task of the program is ready for execution immediately after a signal is received from the coordinator. The signals reporting a task completion should be further processed. When task 1 is completed, the model shows (transition a_{59}) that task 2 and task 3 may start. They are succes-

sively sent in location BP13 (task ready). From there, through connector 31, the ready tasks are transmitted to the PPM which will introduce them in queue b_4 (see Fig. 5.2). The last task of the program, task 4, can only start when task 2 and task 3 are completed, as transition a_{61} shows.

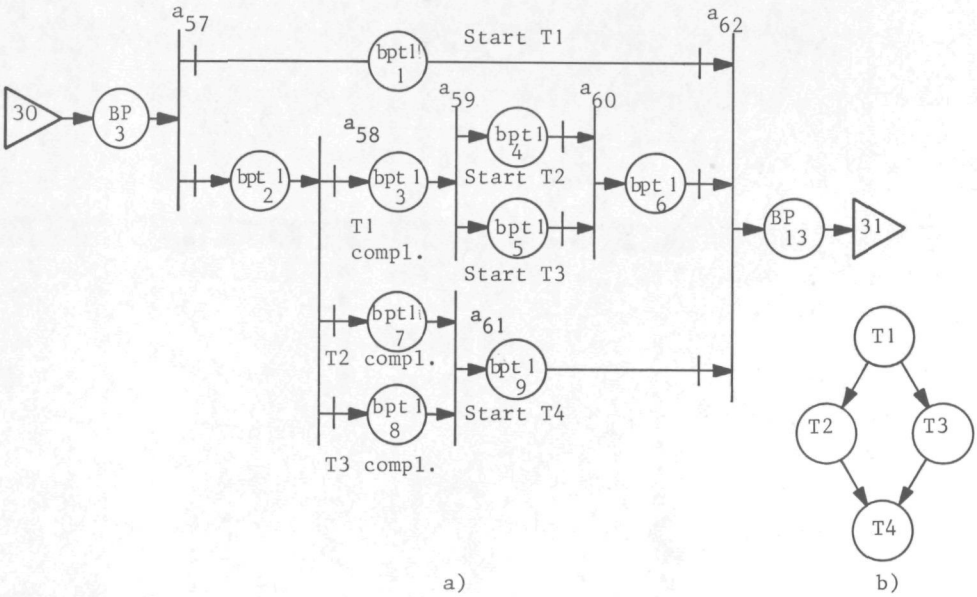


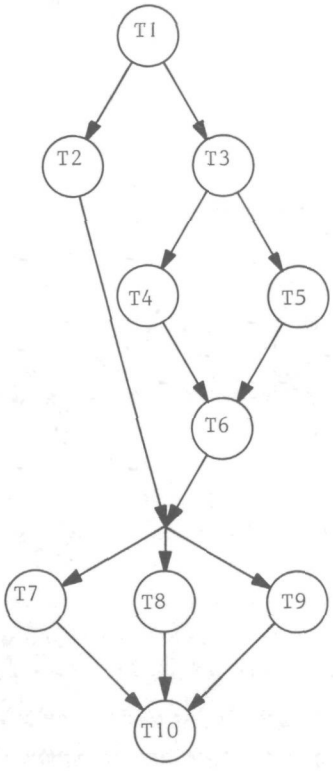
Fig. 5.3. Example of a parallel program pattern.

For the other three parallel program patterns used during the simulation experiments reported in the next section, the relationship between tasks is indicated in Fig. 5.4.

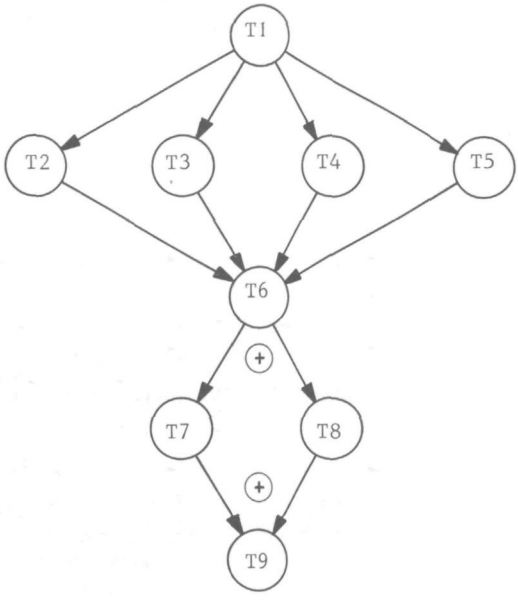
The following parameters have been chosen to characterize a job:

- time of arrival in the system;
- nature of the job (terminal, simple batch, parallel program);
- average execution time between I/O requests;
- number of I/O requests;
- identifier of the assigned PCB.

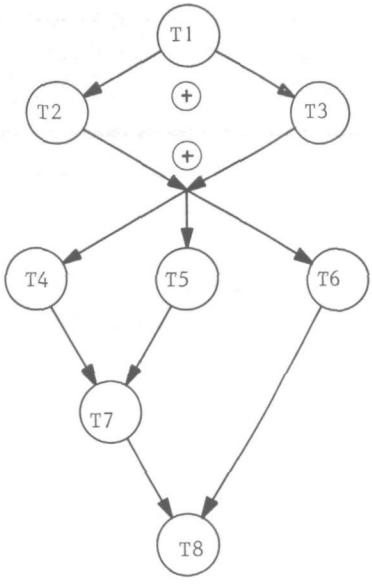
Supplementary parameters are used for the specific needs of the terminal requests and of the parallel jobs. For instance, a terminal request



a)



b)



c)

Fig. 5.4. Parallel programm patterns used in simulation.

carries an identifier of the terminal that has issued the request. A parallel job is provided with an identifier of the coordinator that has been assigned to the job and, due to the limitations of the present model, with an identifier of the pattern chosen for the job. Each parallel task carries an identifier, not only for the job to which it belongs, but also for itself.

The I/O operations related to the reading of a new batch job in the system and to the printing of the job results are always directed to the same disk (unit 0). All other I/O operations are uniformly distributed among the disk units present in a certain configuration.

A terminal issues a request and then waits for the answer. When the answer arrives,it is assumed for the model that the terminal issues the next request after an exponentially distributed "thinking time." The batch jobs arrive at the card readers in accordance with a Poisson distribution (i.e., the intervals between successive arrivals are exponentially distributed). A certain percentage of the batch jobs are parallel programs.

5.3. Simulation experiments and results.

A simulation program for the developed model has been written in the programming language SIMULA 67 [1], [2]. This language has certain features which are related to Concurrent Pascal, and this fact is one of the reasons for its selection.

The writing of the simulation program is simplified by observing that a direct connection can be established between the E-nets concept of "transition" and the SIMULA concept of "class": each transition of the model has been translated as a SIMULA class. The main program must only initialize the system, start the simulation, test for the condition to end the program execution, and print the results.

In the first series of experiments three different scheduling policies have been used. These scheduling policies are described below:

1. Parallel jobs maintain their priority over simple batch jobs in PCB allocation, but only one parallel job at a time can be active.

2. Several parallel jobs can be simultaneously active, but each job has at most one active task at a time.

3. Parallel jobs are executed in multitasking; i.e., the restriction introduced in the second scheduling policy is removed.

The first scheduling policy is very restrictive in the treatment of the batch requests. Once the execution of a parallel job has started, the processing of the batch input queue can only continue until the next parallel job is encoutered. From that moment, no other jobs are taken out of the input queue until the parallel job in execution is completed. The tasks of this job are executed one by one; thus each new task must compete with the terminal requests for a PCB.

The second scheduling policy approaches the situation in a common multiprogramming system. In order to make the comparison with the multitasking scheduling system possible, however, it is assumed that tasks from the parallel jobs are executed independently. In normal multiprogramming, the entire parallel job would be executed as a single "task;" i.e., it would have to compete only once to obtain a PCB.

With tasks executed independently, the second and the third scheduling policies introduce the same amount of system overhead.

During these first experiments, the system configuration has been constant:

- 2 CPUs,
- 4 card readers,
- 15 terminals,
- 2 disk units,
- 4 line printers.

The workload parameters have also remained constant at the values indicated below.

- Proportion of parallel programs: 50%;
- Number of parallel programs patterns: 4;
- Terminal "thinking" time: 30 seconds;
- Batch job interarrival time: 120 seconds;
- Average execution time for I/O requests: 200 msec;
- Time required for printing the results of a job: uniformly destributed between 20 and 90 seconds.

The number of I/O operations and the interval between two successive I/O requests are given in Table 1, for each category of jobs. For the parallel programs, the figures given in Table 1 refer to a task, not to an entire job. It has been assumed that the parallel programs are, in general, more CPU bound than the simple batch jobs, which in turn are more CPU-bound than the terminal requests.

Table 1. The I/O parameters of the workload.

Category of jobs	Number of I/O requests	Interval between requests (msec)
Terminal requests	5	200
Simple batch jobs	40	400
Parallel programs	10	600

Only one coordinator is required by the first scheduling policy. In the other two scheduling policies, 4 coordinators have been used.

For each scheduling policy the experiment consisted of a program run for 5 PCBs, another run 10 PCBs and a third run for 20 PCBs.

The output of a simulation run provides the items of information listed below.

- Histograms of the response (turnaround) time for the job in the three categories;
- Histograms of the job execution time for each job category;
- Histograms of the job waiting time in the CPU queue;
- Data about resource utilization;
- Total simulated time.

A simulation run was stopped after the execution of 80 parallel programs. The recording of the system events necessary in calculating the simulation results was started only after the completion of the first 20 parallel programs. In this way, it may be assumed that the simulation results refer to a system in steady-state operation.

The program was executed on the CDC CYBER 73 and CYBER 173 computers, under the NOS/BE 1.1 operating system, at the *Stichting Academisch Rekencentrum Amsterdam (SARA)*. The NDRE SIMULA compiler was used. A program run required between 400 and 600 seconds of CPU time.

A synopsis of the results obtained is shown in Table 2. Several histograms of the turnaround time and waiting time in the CPU queue are included in Appendix 3.

For the first scheduling policy, the turnaround time of the batch programs has very high values and the utilization of the system resources is low in comparison with the other scheduling policies. As expected, the terminal requests receive a very good service, because there are not many batch

Table 2. Simulation results for a system with 2 CPUs.

Parameters	1 coordinator no multitasking			4 coordinators no multitasking			4 coordinators multitasking		
	5 PCBs	10 PCBs	20 PCBs	5 PCBs	10 PCBs	20 PCBs	5 PCBs	10 PCBs	20 PCBs
Average response time (sec)									
Terminal requests	3.59	3.32	3.48	8.11	6.55	5.82	5.84	5.99	5.47
Simple batch jobs	1090.0	291.0	359.0	173.0	119.0	114.0	105.0	105.0	112.0
Parallel programs	1220.0	411.0	481.0	294.0	208.0	187.0	146.0	149.0	141.0
Average execution time (sec)									
Terminal requests	1.20	1.18	1.18	1.19	1.20	1.19	1.17	1.20	1.18
Simple batch jobs	15.6	16.8	16.4	17.1	15.8	16.2	17.0	15.7	16.3
Parallel programs	48.7	48.1	48.6	48.6	50.5	48.9	45.8	46.4	49.1

Table 2. (contd.) Simulation results for a system with 2 PCUs.

Parameters	1 coordinator no multitasking			4 coordinators no multitasking			4 coordinators multitasking		
	5 PCBs	10 PCBs	20 PCBs	5 PCBs	10 PCBs	20 PCBs	5 PCBs	10 PCBs	20 PCBs
Resource utilization (%)									
CPU 1	72.3	73.6	74.4	92.6	85.9	83.3	80.7	74.6	72.5
CPU 2	72.3	73.3	74.9	92.4	85.5	83.4	80.5	74.6	72.5
Disk 0	28.9	29.7	30.2	34.1	32.2	31.3	31.0	28.7	29.1
Disk 1	28.0	29.0	29.7	30.5	30.8	29.9	29.8	27.8	27.9
LP 1	40.0	36.4	36.5	57.4	50.2	44.7	51.0	45.6	37.8
LP 2	37.2	36.4	37.7	51.9	53.5	44.5	50.1	44.5	40.4
LP 3	38.3	38.1	41.5	57.6	51.2	52.8	47.8	41.8	40.2
LP 4	35.6	41.8	42.8	53.5	51.3	43.3	47.0	40.6	37.1
Throughput in 1 hour									
Terminal requests	1581	1633	1652	1419	1497	1495	1515	1486	1511
Simple batch jobs	50	51	56	64	66	57	73	57	54
Parallel programs	52	52	52	81	71	67	60	56	50

requests to compete for the available PCBs. The system throughput increased slowly with the number of PCBs. The turnaround time is improved when 10 PCBs are used instead of 5 PCBs, but becomes worse in a system with 20 PCBs. Therefore, this first scheduling policy might be recommended for a system used primarily for time-sharing processing, but it is, as expected, not suitable for the system studied in this thesis.

An aspect worth noting in a comparison between the results obtained with the second and the third scheduling policies is the substantial improvement in the turnaround time of the parallel programs when multitasking is used. This improvement is obtained, however, at the expense of a lower resource utilization and, thus, a lower throughput. Therefore, the second scheduling policy is recommendable in a system with the configuration and workload as specified here, when the throughput is the main performance objective. On the other hand, multitasking should be used if the most important requirement is the turnaround time and some decrease in the throughput can be accepted.

A further examination of the results obtained when multitasking is used shows that the general system performance is reduced if the number of PCBs increases. This can be explained by the increase of the time that a PCB must wait in a CPU queue, when the ratio between the number of PCBs and the number of CPUs is too high (see the histograms of the waiting time, included in Appendix 3). The probability that tasks of the same parallel program will be executed simultaneously in such a system is low because too many tasks are allowed to compete for an available CPU.

In the second scheduling policy, the response time for the parallel programs is noticeably improved when the number of PCBs increases, although it remains worse than the response time obtained with multitasking. A possible explanation for the improvement is that the probability of a PCB being available, when a task is ready for execution, increases with the number of PCBs. Moreover, there are no tasks of the same parallel program to compete with each other.The histograms of the waiting time for a CPU, included in Appendix 3, show indeed that the average waiting time is shorter for this scheduling policy than for multitasking.

A supplementary series of experiments was performed in order to study the effect of additional CPUs on the results obtained with the second and the third scheduling policies. In these experiments, the number of PCBs

Table 3. Simulation results for 10 PCBs and variable number of CPUs.

Parameter	4 CPUs		6 CPUs	
	2nd policy	3rd policy	2nd policy	3rd policy
Average response time (sec)				
Terminal requests	2.37	2.46	2.20	2.29
Simple batch jobs	84.6	81.8	83.20	82.50
Parallel programs	124.0	105.0	121.0	99.7
Average execution time (sec)				
Terminal requests	1.18	1.18	1.17	1.20
Simple batch jobs	15.8	16.6	16.5	16.9
Parallel programs	45.7	49.5	49.6	48.5
Resource utilization (%)				
CPU 1	38.8	44.2	25.2	26.1
CPU 2	38.5	43.6	25.5	26.1
CPU 3	38.9	44.3	26.0	25.7
CPU 4	38.9	44.0	25.1	25.5
CPU 5	-	-	25.1	26.2
CPU 6	-	-	25.1	26.5
Disk 0	31.3	33.9	30.4	31.8
Disk 1	30.7	33.3	30.1	30.0
LP 1	37.0	79.4	38.0	39.6
LP 2	43.1	46.3	39.5	42.8
LP 3	44.0	50.6	40.0	40.6
LP 4	42.5	49.3	39.7	43.7
Throughput in one hour				
Terminal requests	1692	1753	1694	1670
Simple batch jobs	53	66	54	59
Parallel programs	57	67	53	53

remained constant at a value of 10. For each policy a configuration with 4 CPUs and a configuration with 6 CPUs were simulated. The other characteristics of the system (equipment and workload) were maintained at the same values as in the previous experiments. The results obtained are presented in Table 3.

The most important fact indicated by these results is that, for the considered configurations, multitasking is superior from all points of view. However, the used workload is insufficient for a system with 6 CPUs, and therefore, the results concerning such a system are less significant than those for a system with 4 CPUs.

In conclusion, when the turnaround time of certain programs must be reduced, multitasking is recommendable. For some system configuration and workload parameters, multitasking can also produce an improvement of the system throughput in comparison with other scheduling policies. But, in general, a reduction of the turnaround time is accompanied by a reduction of the throughput. The relation between the workload parameters, system configuration and scheduling policy is very complex, and each case must be, perhaps, individually studied.

References.

[1] Birtwistle, G.M., et al., SIMULA Begin, Studentlitteratur, Lund, 1973/ Petrocelli Charter, New York, 1974.

[2] Dahl, O.J., et al., SIMULA Information,Common Base Language, Norwegian Computing Center, Oslo, 1970.

[3] Noe, J.D. and Nutt, G.J., "Macro E-nets for Representation of Parallel Systems," IEEE Tr.on C., Vol. 22, No.8 (Aug.1973) pp. 718-727.

[4] Sherman, S., et al., "Trace-Driven Modelling and Analysis of CPU Scheduling in a Multiprogramming System," CACM, Vol.15, No. 12 (Dec. 1972), pp. 1063-1069.

Chapter 6

CONCLUDING REMARKS

After a brief review of the system proposed in the thesis, this chapter contains a few comments and indicates some problems which, in the author's opinion, deserve further investigation.

6.1. Review of the system structure.

In the previous chapters, the characteristics of the proposed computer system have been gradually introduced, as needed for the implementation of the design decisions stated at the beginning of Chapter 3. Here, a complete description of the system, as it has emerged from the design, is given. At the same time, a few more details are given about the structure of the I/O section.

The system considered in this thesis is a multiprocessor system, consisting of a number of identical CPUs connected to a common main memory by means of an interconnection scheme whose structure has not been detailed. The main memory is divided into separately addressable blocks (modules). Specialized, programmable I/O processors, each capable of independently accessing the main memory through the same interconnection scheme as the CPUs, control the I/O section. Each peripheral device is permanently connected to a certain I/O processor and, therefore, can be operated upon only by that processor. The connection between a peripheral device or a group of peripheral devices of the same type is implemented as a "logical channel," i.e.,

an independent control and data path. In this way, an I/O processor can have several simultaneously active devices, one for each logical channel, provided that the access from the I/O processor to the main memory is fast enough to accomodate the information implied in the transfer on all logical channels.

The computer uses a virtual memory. For each process, the virtual memory consists theoretically of the entire information known to the file system. The virtual memory is divided into segments, each segment containing one or more pages. It has been assumed that the page size is 128 memory words and that a segment cannot have more than 32 pages. The page frame (i.e., an area of the size of a page) is the unit for main memory allocation. The pages of a certain segment may occupy non-contiguous page frames in the main memory.

It has been assumed that the operating system should be written in Concurrent Pascal. In this case, the operating system consists of a set of cooperating processes which create an environment for combined batch and time-sharing processing. A specific feature of the system is its ability to concurrently execute tasks of the same program (multitasking). This produces a shorter turnaround time for the programs divided into tasks (such programs are denoted as "parallel programs").

A special hardware device - Bus Master - is provided for the allocation of the CPUs to the processes ready to execute. The Bus Master is connected to the CPUs and to the I/O processors through three buses: an interrupt request bus, a current priority bus and a master request bus. This device ensures that the processors are always allocated to the ready processes having the highest priority levels. The priority of a process, being primarily determined by the function currently performed by the process, may vary during the process execution.

The language Concurrent Pascal requires a few primitive operations in order to implement the monitor concept used for process coordination. These primitive operations are provided in the system as kernel routines whose code contains several special instructions. The basic one among these instructions is "Test and Set" (TST) which allows a CPU to read, examine and, if required, modify the contents of a memory location, without interference from other processors.

Other instructions are imposed by the presence of the Bus Master. They allow a processor to inform the Bus Master about the modification of the

priority of a process, to validate and invalidate an interrupt level and to issue an interrupt request at the appropriate priority level.

For the operating system in this design, the number of processors (CPUs) in a certain instalation is irrelevant. The kernel ensures that the available processors, all having the same architecture, are correctly used. From another point of view, one can say that there is no privileged processor, because all can execute anyone of the functions required in the operating system.

Another major objective in the system design has been the sharing of information. This is implemented by using a capability based addressing scheme and by allowing all processes that use a segment to share a single copy of that segment in the main memory. The segment is thus the unit of information sharing and protection. A capability is a unique identifier of a segment: during the entire existence of the system, once a certain capability is allocated to a segment (when the segment is created), it is not used again to identify another segment.

The protection problem is solved by organizing the segments into program packages and specifying the access rights to the packages and to the individual segments almost entirely during compilation. Inside a package, the access rights are completely established by the compiler. For outside calls, the compiler ensures that a standard procedure is obeyed, which allows a complete protection between the execution environments of the calling and of the called packages.

In order to increase the instruction execution speed, each CPU is equipped with a set of Address Translation Registers (ATRs) which keep the information required for a direct access to the memory pages most recently used by a process. Unlike other implementations based on capability addressing, the approach taken here has been to provide for an automatic manipulation of the contents of these registers. This approach is, in fact, imposed by the use of a page as the unit of main memory allocation, but not as a unit of information separately identifiable by the programmers. The CPUs must have at least two special instructions related to the ATRs: one to save the contents of the ATRs in a process stack when a processor is de-allocated and the other to restore the ATRs when the processor resumes a process. It is also required that two addressing modes be used: real addresses and virtual addresses. There are some operations, like those involved in processing a page fault (i.e., the need to address a page which

does not have a descriptor in an ATR), which can be executed only by using real addresses.

The structure of an I/O processor has not been specified to the same level of detail as the structure of a CPU. An I/O processor can execute channel programs which always use real memory addresses. A component of the operating system, the core monitor, ensures that the memory pages involved in an I/O transfer are not reallocated before the transfer is completed.

A channel program must be prepared by a CPU,which is also responsible for the start of the I/O operation when the logical channel to the requested device becomes available. An I/O processor must be able to report, by means of an interrupt signal, the completion of an I/O transfer. Afterwards, the Bus Master takes care that the appropriate I/O process is activated in order to analyze the effect of the transfer completion.

The I/O processor is also able to examine a set of fixed location in the main memory, one location for each logical channel, from which it obtains the starting address of a new channel program.

6.2. Conclusions.

The work reported in this thesis indicates a possible way of specifying and designing a multiprocessor system. Its main purpose has been to identify the specific problems which appear in the multiprocessor systems, to analyze these problems and to suggest possible solutions and useful tools for the implementation of these solutions.

A hypothetical system has been used to illustrate the discussed problems and to introduce, when required, some details. However, the thesis does not contain the complete specifications of the system, nor a detailed design of all the specified parts.

A simulation model has been developed in order to study the usefulness of a particular system feature, namely, multitasking. The model represents the main features of the system described so far, but contains a number of simplifications for practical reasons. The model has indicated a substantial improvement in the turnaround time of the programs using multitasking at the expense of a reduction of the system throughput, at least in some system configurations.

6.3. Directions for further work.

Some of the general problems concerning the multiprocessor systems which have been identified during the work for this thesis are listed below. (Part of these problems have been treated in various levels of detail in the thesis).

The most important hardware problems can be stated as follows:

- To provide a suitable processor-memory interconnection.
- To provide suitable machine instructions for implementating process coordination.
- To find ways in which functions, normally performed by the operating system, can be advantageously implemented in hardware, in order to increase the computer performance.

The basic software problems are now formulated:

- To design an efficient higher level programming language for the implementation of operating systems (Such a language must contain powerful means of expressing the process coordination and must enable the designer to produce operating systems with a high degree of reliability).
- To provide the methods and the service routines for system recovery and reconfiguration in case of failure.
- To ensure a flexible service for the user jobs.
- To provide flexible methods for information sharing between the users and between the processors.

From the beginning, in fact, the system design process should not be separated into hardware design and software design. Only after analyzing the global system requirements, can the division of the system functions between hardware and software be decided. In this first stage of the design, a system model is a valuable tool. The model should be hierarchically built, in order to allow a study of the system at each required level of detail.

A multiprocessor system with processors having an identical architecture, as considered in this thesis,has the advantage of an improved reliability and allows a graceful degradation. This is obtained because each processor is able to execute any of the functions required by the system.

It has also been assumed in the thesis that the process-memory interconnection scheme gives no preference to a certain processor for a certain memory module. Separate address mapping devices are provided in each computational processor.

In some recent studies [5] of multiprocessor systems based on LSI microprocessors, each processor has fast access to a local memory. When a processor needs information from another processor's local memory, this is obtained with some delay through a mapping scheme used in common by all processors.

Multiple copies of a piece of information (e.g., a memory page) may exist in such a system. When one of the copies is modified by a processor, all other copies should be accessed and updated accordingly. This creates supplementary problems.

It would be of interest to study different implementations for the processor-memory interconnection scheme and for the mapping scheme used with microprocessors and to compare the performances that can be obtained with the two approaches. This is considered as one of the areas deserving further study. It is certain that the second approach introduces some new requirements for the system software. It is preferable that the information currently addressed by a processor should be placed in the processor's local memory in order to reduce the mapping delays.

Capability addressing is preferable to other addressing methods when a flexible information sharing and protection is an important system requirement. Therefore, it seems to be the most suitable addressing method for multiprocessor systems. The capability implementation proposed in this thesis is only one of many possibilities. Several other implementations are described in the literature. For this reason, a second direction in which further work can be done is the study of alternative capability implementations, perhaps with more precise protection specifications. A challenging problem [4] is to design a system which will allow a segment to contain both data and capabilities.

The concepts of process, monitor and class, introduced in the programming language Concurrent Pascal, are very useful tools for a systematic specification of the structure of an operating system. The implementation of these concepts is facilitated by using capability addressing.

Concurrent Pascal introduces, however, a few limitations, for example, the strict type definitions and a fixed system configuration (invariable number of processes). This programming language is, in fact, still in development and, to this author's knowledge, there is only one complete implementation of a system written in Concurrent Pascal. The available literature about this system [1], [2], [3] shows that the designer has extended, during the work, the previously known definitions of some of the concepts.

No proposals for other extensions can be made here. Some experimental designs of simple operating systems using Concurrent Pascal as the implementation language are one way of acquiring the necessary practical knowledge about the language limitations and desirable extensions.

Finally, a few words will be said about the system modelling and simulation. Writing the simulation program and the operating system in the same programming language, or in related languages, has certainly a methodical advantage: changes in the operating system can be easily translated into chages in the simulation program and vice versa. This seems to be the case with an operating system written in Concurrent Pascal and a simulation program written in SIMULA.

The simulation model developed in Chapter 5 must only be considered as a first attempt and does not provide enough information to derive definite conclusions about the usefulness of multitasking and, particularly, about the relation between the turnaround time and the system throughput. A new version of the model, which takes into account the system overhead and the memory limitations should be developed as the first step in extending the scope of the work reported in this thesis.

References

1. Brinch Hansen, P., "The Solo Operating System: A Concurrent Pascal Program," Software-Practice and Experience, Vol. 6, (April-June 1976,) pp. 141-149

2. Brinch Hansen, P., "The Solo Operating System: Processes, Monitors and Classes," Software-Practice and Experience, Vol. 6 (April-June 1976,) pp. 165-200.

3. Brinch Hansen P., "The Solo Operating System: Job Interface," Software-Practice and Experience, Vol.6, (April-June 1976,) pp. 151-164

4. Cosserat, D.C., "A Data Model Based on the Capability Protection Mechanism," R.A.I.R.O,. Vol.9 (Sept. 1975), B-3, pp. 63-68.

5. Swan, R.J., et al., "The Architecture of Cm*: a Modular, Multi - Microprocessor," Carnegie Mellon University, Aug. 1976

Appendix 1

INTRODUCTION TO CONCURRENT PASCAL

The programming language Concurrent Pascal has been proposed by P. Brinch Hansen as a language for operating system implementation [1]. It extends the sequential programming language Pascal and also uses some concepts of SIMULA.

An essential feature preserved from Pascal is the statement which allows a programmer to define explicitly data types. The information in the type definitions is used by a compiler to check for the consistency of a program. In this way, a more extensive verification of program correctness can be performed during compilation, which improves the "reliability" of a large program.

The global data structure of an operating system can be divided into units of closely related data. A set of operations (procedures) is then associated whith each such unit. The combination of one of these data units with its associated operations is denoted an abstract data type. An operating system can be designed in such a way that a certain data structure is always accessed by calling one of its associated procedures. This rule can be, in principle, enforced by a compiler. The procedures associated with a data structure are then the only means of accessing that data structure. Other components of the operating system need only know what operations may be performed on the data structure, but can ignore the details of carrying out these operations.

To make such a system organization possible, Concurrent Pascal introduces three generic abstract data types: processes, monitors and classes. Program variables declared to belong to one of these generic types are called system components. An entire system will appear as a Concurrent Pascal program.

A process consists of a set of private data, a (set of) sequential program(s) and a set of access rights to other system components (Fig. A1.1.a). One process cannot operate on the private data of another process, but concurrent processes can share certain data structures by means of monitors. A system component of the type "process" is an active component; i.e., it is responsible for starting and conducting activities required in the operation of the system. A system can be, therefore, imagined as a collection of processes operating simultaneously (concurrently).

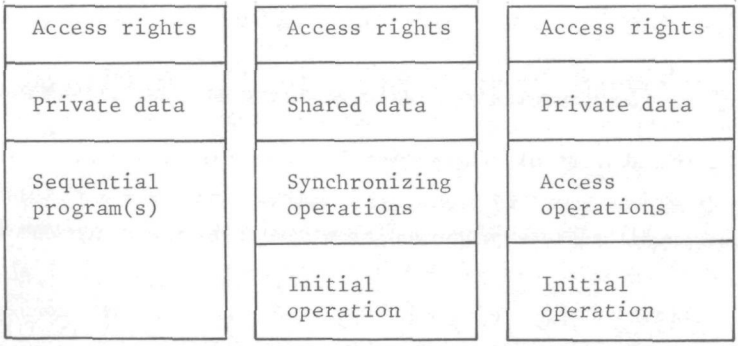


Fig. A1.1. Abstract data types defined in Concurrent Pascal.

A monitor (Fig. A1.1.b) defines a shared data structure and all the operations that individual processes may perform on it. These operations are called monitor procedures. A monitor also defines an initial operation executed only once, when the monitor data structure is created. A monitor can have access to other system components of the type "monitor" or "class".

If several concurrent processes simultaneously call procedures of the same monitor, these procedures must be executed strictly one at a time. While operating, a monitor procedure has, thus, exclusive access to the monitor data. The machine that runs an operating system written in Concurrent Pascal must be able to delay processes for short periods of time, until they can enter a monitor.

A process may find, while executing a monitor procedure, that an expected condition is not satisfied. In such a case, the process must be delayed in an internal monitor queue and the monitor can be entered by other processes.

A primitive operation, "delay," is defined for this purpose. Another process will, eventually, fulfill the condition expected by the previously delayed process. The delayed process is then resumed with the aid of another primitive operation, "continue."

A class (Fig. A1.1.c) defines a set of private data and operations allowed on these data. It has a structure similar to that of a monitor, but the class procedures are not implemented to have exclusive access to the data. This is not necessary because the design of a system must ensure that simultaneous calls to the procedures of the same class cannot occur. No provisions for delaying a process inside a class are required.

The design of a simplified spooling system is outlined below. This will serve to illustrate the use and the notations of Concurrent Pascal.

The spooling system considered here accepts input from a card reader and transmits the card images to a disk unit. From the disk unit, the information is accessed and updated, and then the results are transmitted from the disk to a line printer which constitutes the output device of the spooling system.

Three activities which may proceed concurrently can be identified in this system:

- reading cards and creating an "input queue";
- updating the data of the input queue and creating with the results an "output queue";
- printing the contents of the output queue.

Each of these activities will be defined as a separate process type. The system will need one component of each type. Henceforth, the three process types will be referred to as "input process," "job process" and "output process," respectively.

The input process must have access to the card reader and to the input queue on the disk unit. It shares the information in the input queue with a job process. Therefore, the access to the input queue must be controlled by a monitor of the type "disk buffer."

The job process must be able to access both the input queue and the output queue on the disk unit. The need of a monitor for the input queue has already been discussed. The output queue is shared between the job process and the output process and for this reason a monitor is also required to control the access to the output queue.

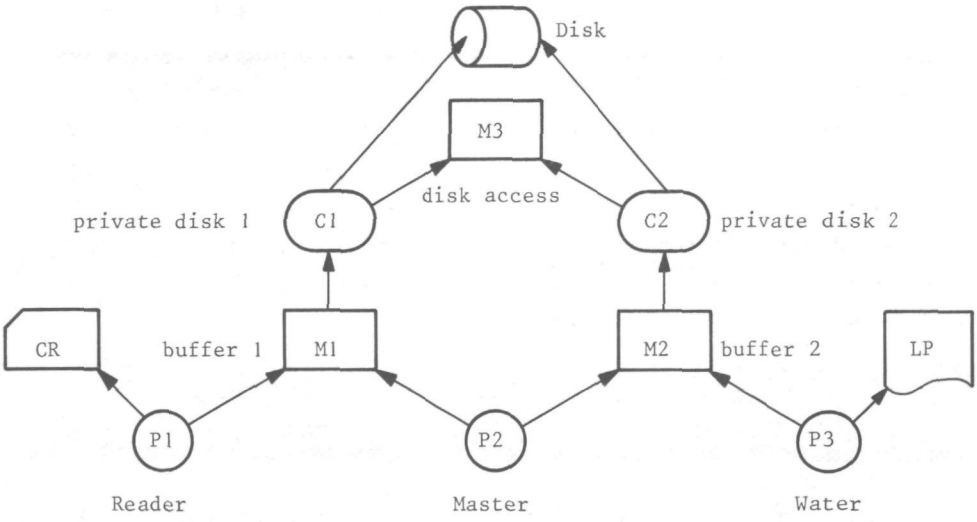


Fig. A1.2. A simple spooling system.

Finally, the output process must have access to the output queue and to the line printer.

The monitors for controlling the access to the two disk queues perform identical functions. With the assumption that in both cases the quantity of information implied in a single disk transfer is the same, one type of monitor will be sufficient. This type is denoted as "disk buffer" and the system must declare two components of the type "disk buffer."

Each disk buffer can be written as if it had its private disk unit. Therefore, a class, "virtual disk," is introduced to describe the data organization on the disk and the disk read and disk write operations. Again, there may be an unique type definition, but the system will declare two components of this type, one in each disk buffer.

The fact that the system has only one disk unit must be made apparent in the definition of the virtual disk class. This class is given access to a monitor, "resource," which takes care that only one class at a time is allowed to perform an operation on the existing disk unit. The system will declare one component of the type "resource."

The relationship between the components of the system whose functional design was given above can be represented graphically as shown in Fig. A1.2. In this figure, process components are depicted as circles, monitors as

rectangles and classes as ovoids. The card reader, the disk and the line printer are represented by the usual symbols. Arrows are used to indicate the access rights of each component. Note the differences between the names of the components represented in Fig. A1.2. and the names of the types to which these components belong. A component is a variable of a certain type. The different names have been selected with the purpose of emphasizing this distinction. The private disk components are shown to have direct access to the real disk unit. In the code of the virtual disk class, however, such an access is always preceded by a call to the resource monitor. Therefore, when the actual access of performed, the disk is certainly allocated to that class. At the completion of a transfer, the resource monitor is called again, to release the disk and, possibly, to allocate it to a waiting process.

At this point, the functional design of the spooling system is completed. The coding of the different system types can now start.

Two approaches may be used in writing the code: to follow the same order used in the functional design (processes-disk buffers-virtual disk-resource) or to follow the reverse order. The reverse order might have the advantage that it does not use any types that are not yet defined. For this reason, the second approach is selected and the coding of the resource monitor is the first to be discussed.

The resource monitor has two procedures: "Request" and "release." The procedure "request" allocates the disk in FIFO order to the incoming requests. When a request cannot be satisfied, the process that has issued the request is delayed in an internal monitor queue, "q." The procedure "release" deallocates the disk from the process that has used it and continues the first process waiting in the monitor queue. If the monitor queue is empty, the disk becomes free.

The code of the "resource" monitor follows.

```
type resource = monitor;
  var free: boolean ; head, tail, length: integer;
      q: array [.0..2.] of queue;
  procedure entry request;
      var arrival: integer;
      begin if free then free: = false else
          begin arrival: = tail;
```

```
                tail:= (tail+1) mod 3;
                length:= length+1;
                delay (q(.arrival.));
              end;
            end;
          procedure entry release;
            var departure: integer;
            begin if length=0 then free: = true else
              begin departure: = head
                head: = (head+1) mod 3;
                length: = length-1;
                continue (q(.departure.));
              end;
            end;
          "initial statement"
          begin free: = true; length:=0; head:=0; tail:=0; end;
```

There are three internal queues in this monitor. It may indeed happen that all three processes of the system must wait in the "resource" monitor until a previous disk operation is completed.

Notice that the primitive operations "delay" and "continue" should only specify the queue element implied in the operation. It is the responsibility of the virtual machine that implements Concurrent Pascal to know the identity of the affected process and to save (or restore) the necessary information about the process status.

The input/output (I/O) operations are performed in Concurrent Pascal under the control of a standard procedure, "io":

```
io (block, param, device)
```

where

"block" is a main memory buffer which contains or receives the transferred data;

"param" is a variable of type record;

The record "param" has the structure indicated below.

```
var param: record
             operation : iooperation;
             result    : ioresult;
             pageno    : integer;
           end;
```

"device" is an identifier of the requested peripheral unit. The type "iooperation" specifies the possible kinds of I/O operations: read, write, rewind, seek, etc. The type "ioresult" specifies the manner in which an I/O operation is completed: normal, transmission error, device error, etc. The component "pageno" in the variable "param" indicates, for a disk, the address of the disk page referenced in the transfer.

With these details about the I/O operations, it is possible to write the code for the "virtual disk" class. This class must have access to a monitor of type "resource" and provides two procedure entries: "read" and "write." It can be coded as follows.

```
type virtualdisk = class (diskaccess: resource);
    const disk =?; "the physical address of the real disk"
    var param = record
                    operation : iooperation
                    result    : ioresult
                    pageno    : integer;   end;
    procedure entry read (pageint : integer; var block:page);
        begin with param do
                begin operation:= read;
                      pageno: = pageint;
                end;
            diskaccess.request;
            io (block, param,disk);
            diskaccess.release;
        end;
procedure entry write (pageint:integer; block:page);
    begin with param do
            begin operation: = write;
                  pageno   : = pageint;
            end;
            diskaccess.request ;
            io (block,param,disk);
            diskaccess.release ;
        end;
begin    end;
```

Both procedures require two parameters: "pageint," the address of a disk page and "block," a page of the main memory. The qualifier var for the parameter "block" in the "read" procedure indicates that the contents of this page will be changed. No initial operation is indicated in this class because it does not have any permanent variables (i.e., variables declared before the procedure definitions).

A "disk buffer" monitor must have access to a "resource" monitor, in order to transmit this access to the "virtual disk" class declared in the disk buffer. Two other parameters, "base" and "limit" are used to establish the buffer zone on the real disk. The disk buffer provides two procedure entries:

- "send," used in writing a page to the disk,
- "receive," used in reading a page from the disk.

The code of the disk buffer monitor is given below.

```
type diskbuffer = monitor (diskaccess:resource;base,limit:integer);
   var disk: virtualdisk; sender, receiver:queue;
       head, tail, length: integer;
   procedure entry send (block:page);
       begin if length = limit then delay (sender);
           disk.write (base+tail, block);
           tail:=(tail+1) mod limit;
           length:= length+1;
           continue (receiver);
       end;
   procedure entry receive (var block: page);
       begin if length = 0 then delay (receiver);
           disk.read (base+head, block);
           head:= (head+1) mod limit;
           length:= length+1;
           continue (sender);
       end;
   begin init disk (diskaccess);
       head:=0; tail:=0; length:=0;
   end;
```

A disk buffer monitor has the following permanent variables:

- "disk," a class of the type "virtualdisk."
- "sender" and "receiver," two internal queues.
- "head," "tail" and "length," used to indicate the status of the buffer space.

A process which requests a write operation to the disk (procedure "send") is delayed in the queue "sender" if the buffer space on the disk is full. A process is delayed in the queue "receiver" during an attempt to read while the buffer is empty.

The disk buffer monitor must initialize, during its initial operation, the class variable "disk." A standard procedure, "init," is used for this purpose. The "init" procedure is part of the execution environment of Concurrent Pascal, but it should not be incorporated in the kernel of the operating system. The "init" procedure may use, however, functions provided in the kernel.

A simplified coding of the three types of processes is now presented. For the input process, the code shown below can be used.

```
type inputprocess = process (buffer:diskbuffer);
  var block: page;
  cycle
    readcards (block);
    buffer.send (block);
  end;
```

The statement "readcards (block)" should be interpreted as a call to a program (not shown here) which takes care of the details of reading cards and transmitting the card images to the memory page called "block" until this page is full. The structure cycle . . . end indicates an infinite repetition of the statements contained in its body.

The code of a job process can be written as follows.

```
type jobprocess = process (input, output:diskbuffer);
  var block: page;
  cycle
    input.receive (block);
    update (block);
    output.send (block);
  end;
```

Again, "update (block)" is a call to a program whose details are not given.

The output process is now defined.

```
type outputprocess = process (buffer:diskbuffer);
    var block: page;
    cycle
        buffer.receive (block);
        print (block) ;
    end;
```

The program "print" contains the details of transmitting successive lines of text to the line printer.

The spooling system, like any other system written in Concurrent Pascal, is generated and initialized (started) in a so-called "initial process." This special process consists of three sections:

- the definitions of all system types;
- the declaration of the system components;
- the initialization of the system components.

The initial process must be executed with the aid of a human operator who provides some of the data required during the initialization of the system components.

For this simple spooling system, the initial process is coded as shown below.

```
type resource      = monitor  ... end;
     virtualdisk   = class    ... end;
     diskbuffer    = monitor  ... end;
     inputprocess  = process  ... end;
     jobprocess    = process  ... end;
     outputprocess = process  ... end;

var diskaccess : resource;
    buffer 1, buffer 2: diskbuffer;
    reader: inputprocess;
    master: jobprocess;
    writer: outputprocess;
```

```
begin init diskaccess;
        buffer 1 (diskaccess, base 1, limit 1);
        buffer 2 (diskaccess, base 2, limit 2);
        reader (buffer 1); writer (buffer 2);
        master (buffer 1, buffer 2);

end;
```

Reference

[1] Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Tr. on S. E., Vol.1, No. 2 (July 1975), pp. 199-207.

Appendix 2

FUNDAMENTALS OF THE EVALUATION NETS

The evaluation nets (E-nets) have been proposed and developed [1] by G.J. Nutt and J.D. Noe, as a technique for graphical and formal representation of the structure and behavior of a computer system. Therefore, the E-nets have functions which are similar to those of block diagrams and flowcharts. They are superior, however, in the possibilities offered for the representation of concurrent activities and of the system resources.

The E-nets have evolved from the Petri nets and partially maintain the symbols and the concepts used in the theory of Petri nets.

An E-net contains a set of locations connected over a set of allowable transitions.

Locations represent conditions that can exist in a system for a period of time; they are depicted as circles. Locations represent, thus, the system status.

Transitions represent the activities that take place in the system when certain conditions are satisfied. A transition is graphically represented by a vertical line. Associated with each transition are:

1) a transition schema, denoting the locations related to the transition and their type of connection;
2) a transition time, i.e., the duration of the activity represented by the transition;
3) a transition procedure, describing the effect of the transition on the system status.

Another basic concept is the token, graphically represented as a dot inside a location. The presence of a token in a certain location signifies that the condition represented by that location is satisfied; whereas an empty location means "condition not satisfied." A token may carry attributes (e.g., a token representing a job can specify the job name, memory requirements, CPU time, etc.).

A transition has a set of input locations and a set of output locations. Graphically, the input and the output locations are connected to the transition by arrows. Let us consider a simple transition with one input and output location. When the input location is occupied by a token and the output location is empty, the transition "fires" and after some time the input location becomes empty and the output location, occupied.

If the output location is not empty, the transition cannot fire, notwithstanding the status of the input location. A location cannot hold, thus, more than one token at a time. The transition procedure must specify, for each output location that receives a token, the attributes of that token.

Five primitive transitions have been defined which are represented in Fig. A2.1.

In a T-transition, when a token resides in the input location i and the output location o is empty, the transition fires. After some delay (the transition time), the token is removed from i and appears in o. The transition schema is T(i,o).

The F transition ("fork transition") fires when a token is present in i and o_1 and o_2 are both empty. After firing, the token from i disappears and both o_1 and o_2 receive tokens. The token attribute may be altered by the transition procedure. Moreover, the resulting tokens need not be identical. The transition schema is F (i,o_1,o_2).

The J transition ("join transition") fires when token exists in both i_1 and i_2, but not in o. The tokens are removed from i_1 and i_2, and a token appears in o. The transition procedure specifies the attributes (if any) of the resulting token. The transition schema is $J(i_1,i_2,o)$.

The X transition uses a resolution procedure (incorporated in the transition procedure) to decide whether the token from i should go to o_1 or to o_2. When a token arrives in i, the resolution procedure is activated. If the resolution procedure indicates zero, this means that the path to o_1 should be followed; whereas if it indicates 1, the path to o_2 is selected.

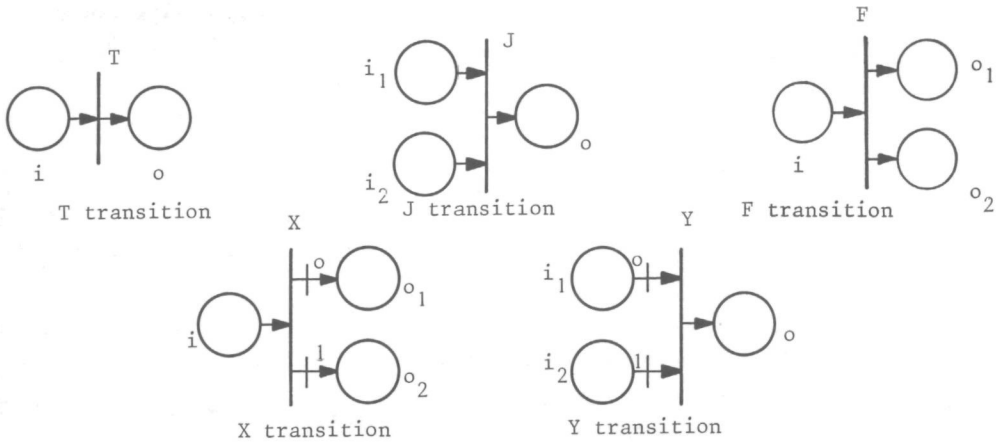


Fig. A2.1. Primitive transitions.

The transition can fire if the selected output location is empty. In the graphical representation of the transition schema, the arrow to the output locations are marked by short vertical bars to suggest that the arrows represent conditional connections. The transition schema is $X(i,o_1,o_2)$.

The Y transition fires when a token appears in one of the input locations, provided the output location is empty. When tokens are simultaneously present in i_1 and i_2, a resolution procedure must be activated to decide which one of the tokens is allowed to pass to the output location. Bars on the arrows from the input locations to the transition indicate conditional paths. The transition schema is $Y(i_1,i_2,o)$.

A resolution procedure has the ability to examine the status of certain locations of the net at the moment of the procedure activation, in order to make a decision about the route of the tokens.

The set of logical relations between the input and output locations contained in the five primitive transitions is sufficient to express most of the inter-component relations encoutered in a computer system.

From the moment that a transition fires until the activity of the transition procedure is completed (thus after a delay specified by the transition time), the status of its input and output locations must be considered as undefined. This is particularly important in the writing

of a simulation program which reproduces the behavior of an E-net.

It often happens in the construction of an E-net model of a computer system that some functions appear in different parts of the model. The model gains in convenience and clarity if the partial E-nets representing these functions are separately defined in terms of the five primitive transitions and then introduced in the general model as macro E-nets. The resulting general model is then a macro E-net model.

A macro E-net used as a transition is called a macro-transition, and one used as a location is called a macro-location. Figure A2.2 shows some of the most useful macro E-nets.

In the X_n macro-transition, a resolution procedure is activated whenever a token arrives in the input location i. This procedure will select one of the n input locations, to which the token should pass. When that location is free, the transition will fire. An X_n macro-transition has, therefore, the function of a multiplexer.

The Y_n macro-transition fires when a token appears in one of the n input locations and the output location is empty. If tokens exist simultaneously in several input locations, a resolution procedure should be activated to decide which of the tokens is allowed to pass.

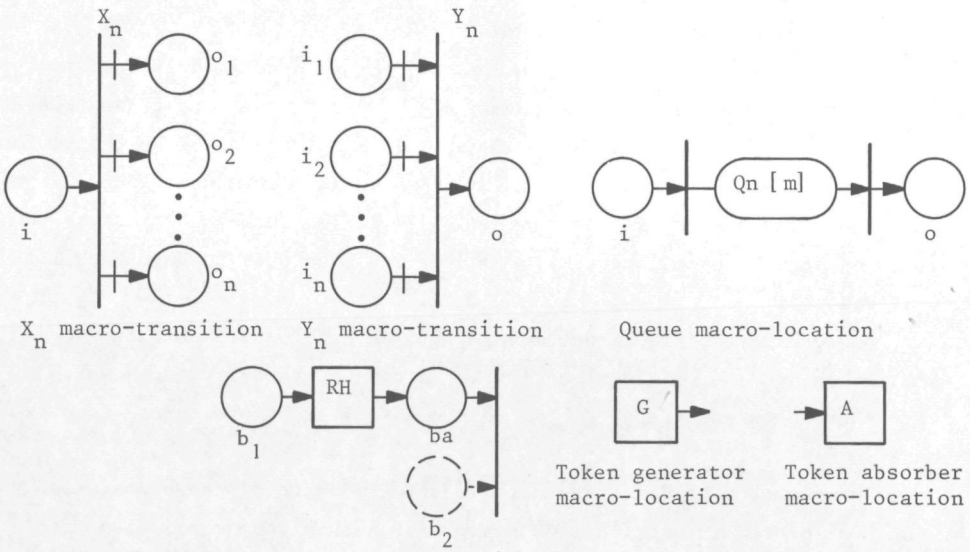


Fig. A2.2. Examples of macro E-nets.

A Y_n macro-transition has the function of a selector.

The usefulness of the X_n and Y_n macro-transitions is especially apparent in representing a set (more than two) of concurrent processes operating in a critical region. The general case is exemplified in Fig. A2.3. At the input of the model, the priority between the simultaneous requests to enter the critical region is decided by a Y_n macro-transition.

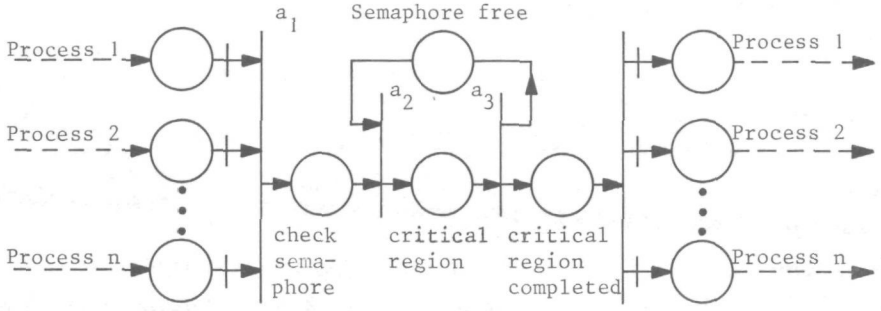


Fig. A2.3. Concurrent process in a critical region.

An X_n macro-transition at the output of the model routes a token denoting the completion of the critical region to the corresponding process, indicated by an attribute of the token.

A queue macro-location appears very often in E-net models of computer systems. The inner logic of the queue may vary from one case to another and must be specified in the separate E-net describing the queue in terms of primitive E-nets. The general model of a queue is always denoted by $Q_n[m]$, where n is the number of locations provided in the queue and m is the number of token attributes. A queue may contain, thus, up to n tokens, each with m attributes.

Because most of the E-net models will be developed as a basis for simulation studies, it is necessary to introduce macro-locations which can generate tokens (denoted by G in Fig. A2.2) or absorb tokens (denoted by A in Fig. A2.2).

The resource handler (RH macro-location) controls a resource which can be allocated in portions. An example of such a resource is the main memory. A request for such a resource is introduced in the location b_q, where the token must have an attribute specifying the size of the request. The location labeled b_a, (assign), receives a token from RH when the request can be satisfied. When quantities of the resource are released, they are

returned to the resource handler through the location b_r (return).

The macro E-nets are suitable for constructing models of computer systems in a hierarchical manner. At each level of the hierarchy, only the details considered as essential for that level are introduced, the other components of the system being replaced by macro E-nets. These macro E-nets can be further detailed in the lower levels of the hierarchy.

Reference

[1] Noe, J.D., Nutt, G.J., "Macro E-nets for Representation of Parallel Systems," IEEE Tr. on C., Vol.22, No. 8 (Aug. 1973), pp.718-727

Appendix 3

HISTOGRAMS OBTAINED FROM SIMULATION

The first nine histograms reproduced in the following pages refer to the response (turnaround) time obtained in a system with 2 CPUs and 10 PCBs, for the terminal requests, single batch jobs and parallel programs, respectively. For each of these categories of jobs, three histograms are presented, corresponding to the three scheduling policies used in simulation.

The last four histograms show the time that parallel programs must wait in a CPU queue in a system with 2 CPUs, when the second and the third scheduling policies are used. The first of these histograms were obtained when 5 PCBs were used and the other two histograms when 10 PCBs were used.

All time values printed in the histograms are expressed in milliseconds. An exponential notation is used. Thus, 3.22" +004 represents 3.22×10^4 milliseconds, or 32.2 seconds. Each histogram is headed by the mean value, standard deviation, minimum value and maximum value of the variable represented in the histogram. The total number of entries is also indicated. The values printed in the column at the left of the histograms are the interval limits. The column at the right of the histograms indicates the number of entries in each interval and, between brackets, their percentage from the total number of entries. The histograms were automatically scaled in relation to the interval containing the maximum number of entries.

HISTOGRAM TERMINAL RESPONSE TIME
MEAN VALUE= 3.32"+003
STD.DEVIATION= 9.76"+004
MIN.VALUE= 6.11"+001
MAX.VALUE= 1.34"+004
NO.OF ENTRIES=1905

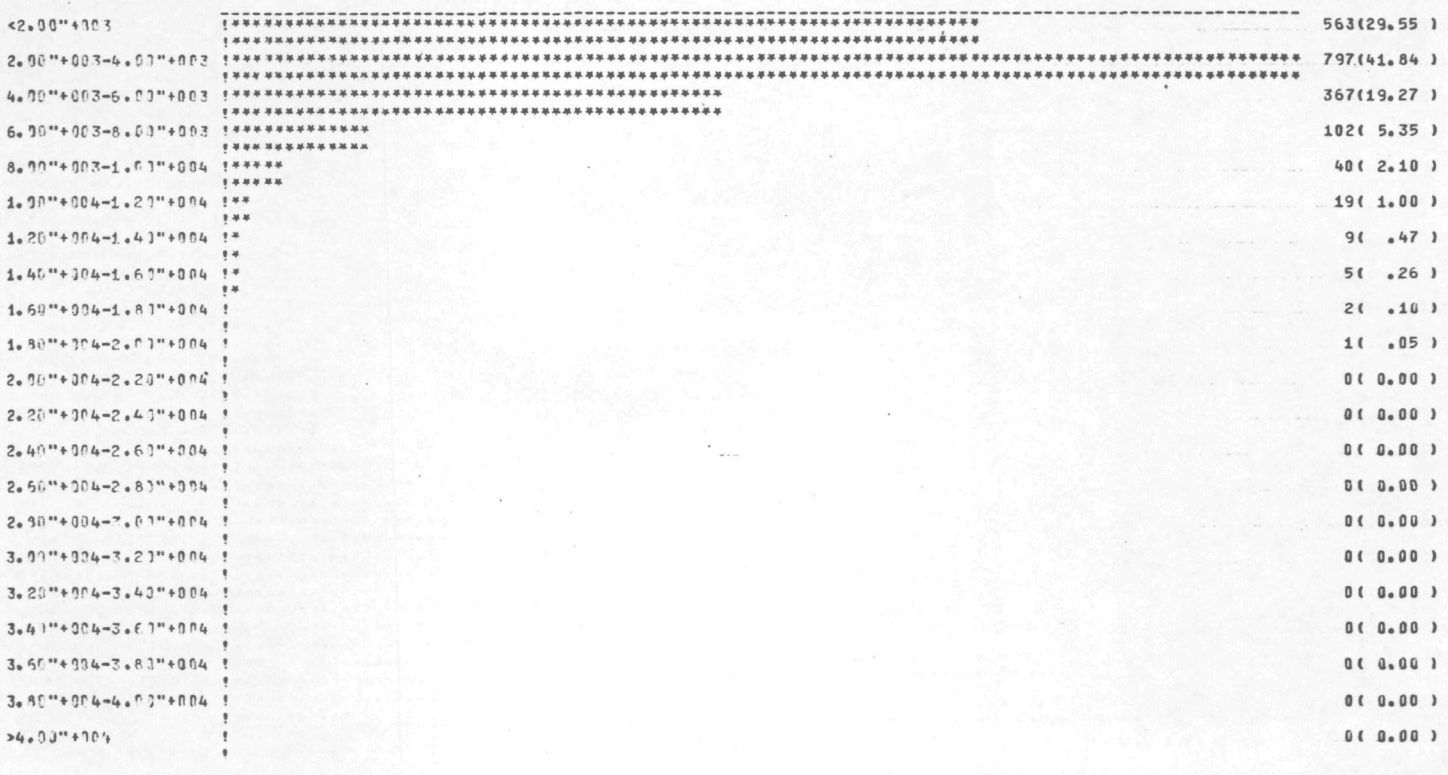


Fig. A3.1. Response time terminal request, first scheduling policy

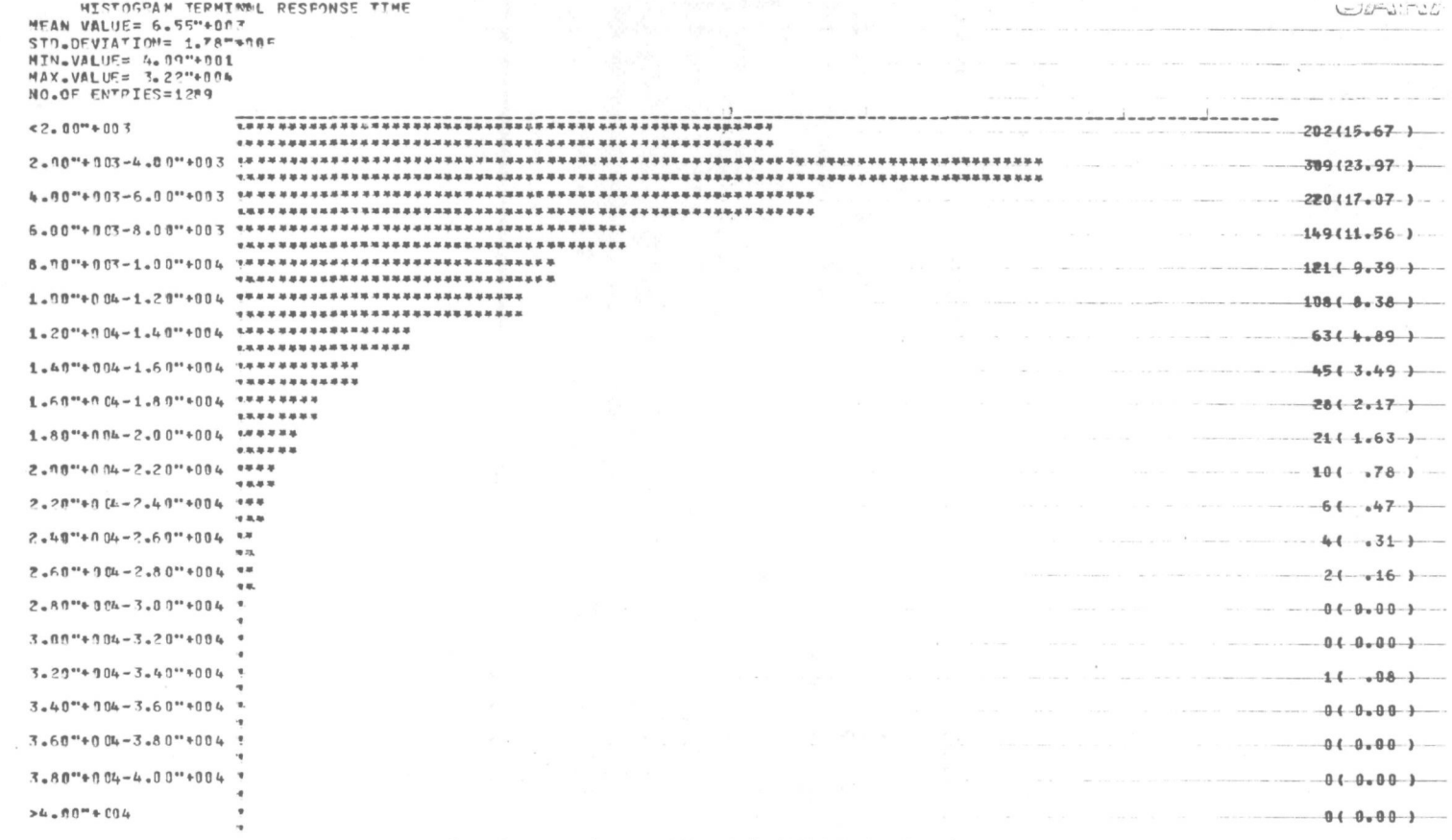


Fig. A3.2. Response time terminal request, second scheduling policy.

```
     HISTOGRAM TERMINAL RESPONSE TIME
MEAN VALUE= 5.99"+003
STD.DEVIATION= 2.10"+003
MIN.VALUE= 6.23"+001
MAX.VALUE= 3.41"+004
NO.OF ENTRIES=1651

<2.00"+003          !*************************************************************************** 359(21.74 )
                    !*************************************************************************
2.00"+003-4.00"+003 !*********************************************************************************************** 467(28.29 )
                    !*********************************************************************************************
4.00"+003-6.00"+003 !****************************************** 210(12.72 )
                    !*****************************************
6.00"+003-8.00"+003 !******************************* 155( 9.39 )
                    !*******************************
8.00"+003-1.00"+004 !************************ 121( 7.33 )
                    !************************
1.00"+004-1.20"+004 !******************** 94( 5.69 )
                    !********************
1.20"+004-1.40"+004 !***************** 87( 5.27 )
                    !*****************
1.40"+004-1.60"+004 !*************** 69( 4.18 )
                    !***************
1.60"+004-1.80"+004 !******* 37( 2.24 )
                    !*******
1.80"+004-2.00"+004 !**** 19( 1.15 )
                    !****
2.00"+004-2.20"+004 !*** 16( .97 )
                    !***
2.20"+004-2.40"+004 !** 8( .48 )
                    !**
2.40"+004-2.60"+004 !* 6( .36 )
                    !*
2.60"+004-2.80"+004 ! 2( .12 )
                    !
2.80"+004-3.00"+004 ! 0( 0.00 )
                    !
3.00"+004-3.20"+004 ! 0( 0.00 )
                    !
3.20"+004-3.40"+004 ! 0( 0.00 )
                    !
3.40"+004-3.60"+004 ! 1( .06 )
                    !
3.60"+004-3.80"+004 ! 0( 0.00 )
                    !
3.80"+004-4.00"+004 ! 0( 0.00 )
                    !
>4.00"+004          ! 0( 0.00 )
                    !
```

Fig. A3.3. Response time terminal request, third scheduling policy.

```
     HISTOGRAM SIMPLE BATCH JOBS RESPONSE TIME
MEAN VALUE= 2.81"+005
STD.DEVIATION= 1.50"+006
MIN.VALUE= 6.48"+004
MAX.VALUE= 7.51"+005
NO.OF ENTRIES=  60

<2.00"+004          !                                                                                                    0( 0.00 )
                    !
2.00"+004-4.00"+004 !                                                                                                    0( 0.00 )
                    !
4.00"+004-6.00"+004 !                                                                                                    0( 0.00 )
                    !
6.00"+004-8.00"+004 !************************                                                                          4( 6.67 )
                    !************************
8.00"+004-1.00"+005 !*****************************                                                                     5( 8.33 )
                    !*****************************
1.00"+005-1.20"+005 !*******************************************                                                         7(11.67 )
                    !*******************************************
1.20"+005-1.40"+005 !************************                                                                          4( 6.67 )
                    !************************
1.40"+005-1.60"+005 !******                                                                                              1( 1.67 )
                    !******
1.60"+005-1.80"+005 !******                                                                                              1( 1.67 )
                    !******
1.80"+005-2.00"+005 !************                                                                                        2( 3.33 )
                    !************
2.00"+005-2.20"+005 !************************                                                                          4( 6.67 )
                    !************************
2.20"+005-2.40"+005 !************                                                                                        2( 3.33 )
                    !************
2.40"+005-2.60"+005 !************************                                                                          4( 6.67 )
                    !************************
2.60"+005-2.80"+005 !                                                                                                    0( 0.00 )
                    !
2.80"+005-3.00"+005 !************                                                                                        2( 3.33 )
                    !************
3.00"+005-3.20"+005 !************                                                                                        2( 3.33 )
                    !************
3.20"+005-3.40"+005 !************                                                                                        2( 3.33 )
                    !************
3.40"+005-3.60"+005 !                                                                                                    0( 0.00 )
                    !
3.60"+005-3.80"+005 !******                                                                                              1( 1.67 )
                    !******
3.80"+005-4.00"+005 !************************                                                                          4( 6.67 )
                    !************************
>4.00"+005          !*********************************************************************************************** 15(25.00 )
                    !***********************************************************************************************
```

Fig. A3.4. Turnaround time batch jobs, first scheduling policy.

```
     HISTOGRAM SIMPLE BATCH JOBS RESPONSE TIME
MEAN VALUE= 1.19"+005
STD.DEVIATION= 7.38"+005
MIN.VALUE= 5.40"+004
MAX.VALUE= 2.22"+005
NO.OF ENTRIES=  57

<2.00"+004          *                                                                                                    0( 0.00 )
                    *
2.00"+004-4.00"+004 *                                                                                                    0( 0.00 )
                    *
4.00"+004-6.00"+004 *******                                                                                              1( 1.75 )
                    *******
6.00"+004-8.00"+004 ****************************************                                                             8(14.04 )
                    ****************************************
8.00"+004-1.00"+005 ********************************************************                                             11(19.30 )
                    ********************************************************
1.00"+005-1.20"+005 ********************************************************************************************     16(28.07 )
                    ********************************************************************************************
1.20"+005-1.40"+005 ****************************************                                                             7(12.28 )
                    ****************************************
1.40"+005-1.60"+005 *****************                                                                                    3( 5.26 )
                    *****************
1.60"+005-1.80"+005 ************************                                                                             4( 7.02 )
                    ************************
1.80"+005-2.00"+005 *****************                                                                                    3( 5.26 )
                    *****************
2.00"+005-2.20"+005 ************                                                                                         2( 3.51 )
                    ************
2.20"+005-2.40"+005 ************                                                                                         2( 3.51 )
                    ************
2.40"+005-2.60"+005 *               0( 0.00 )
                    *
2.60"+005-2.80"+005 *               0( 0.00 )
                    *
2.80"+005-3.00"+005 *               0( 0.00 )
                    *
3.00"+005-3.20"+005 *               0( 0.00 )
                    *
3.20"+005-3.40"+005 *               0( 0.00 )
                    *
3.40"+005-3.60"+005 *               0( 0.00 )
                    *
3.60"+005-3.80"+005 *               0( 0.00 )
                    *
3.80"+005-4.00"+005 *               0( 0.00 )
                    *
>4.00"+005          *               0( 0.00 )
                    *
```

Fig. A3.5 Turnaround time batch jobs, second scheduling policy.

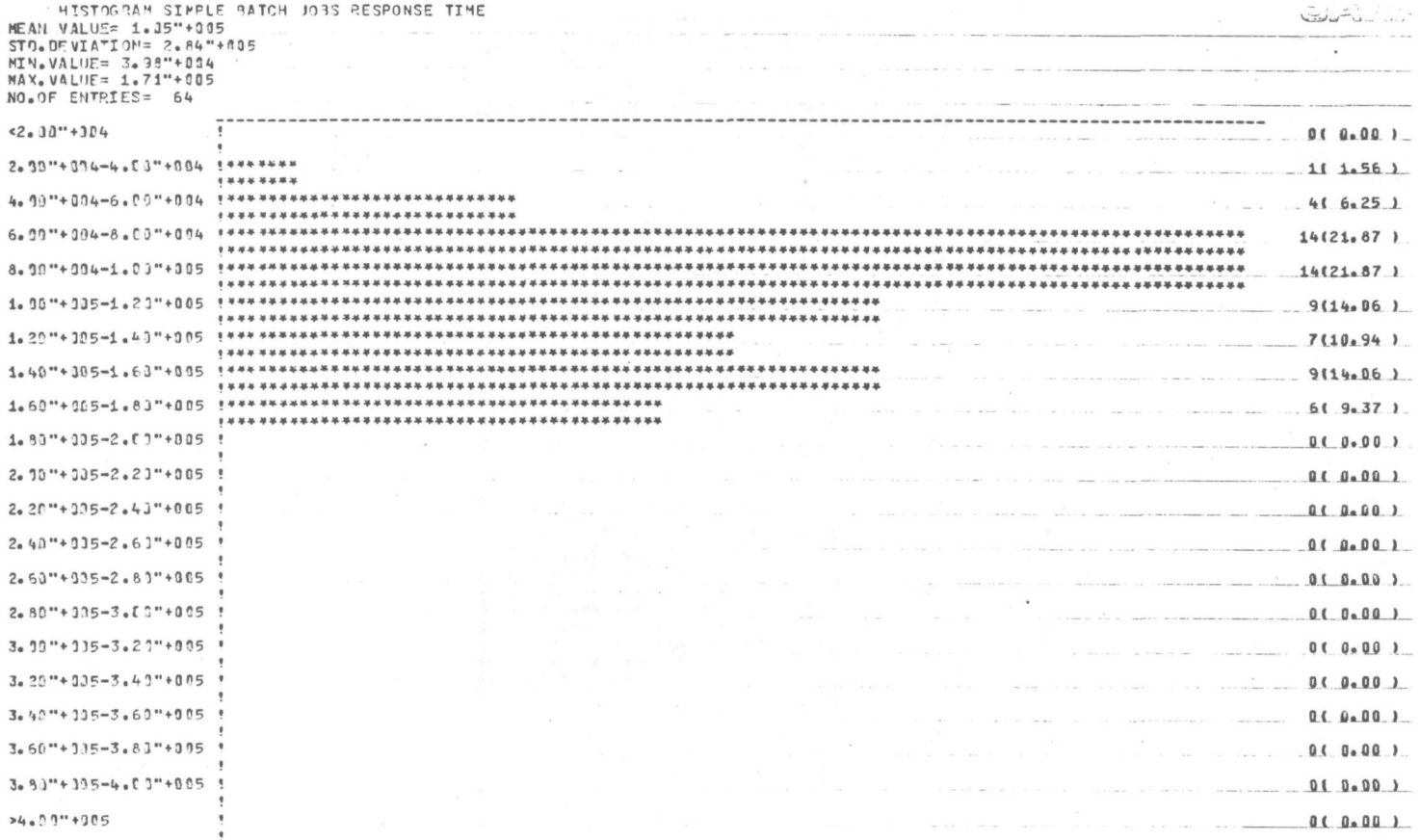


fig. A3.6. Turnaround time batch jobs, third scheduling policy

```
     HISTOGRAM PARALLEL BATCH JOBS RESPONSE TIME
MEAN VALUE= 4.11"+005
STD.DEVIATION= 1.59"+006
MIN.VALUE= 9.76"+004
MAX.VALUE= 8.16"+005
NO.OF ENTRIES=  61

<2.00"+004          !                                                                                                    0( 0.00 )
                    !
2.00"+004-4.00"+004 !                                                                                                    0( 0.00 )
                    !
4.00"+004-6.00"+004 !                                                                                                    0( 0.00 )
                    !
6.00"+004-8.00"+004 !                                                                                                    0( 0.00 )
                    !
8.00"+004-1.00"+005 !***                                                                                                 1( 1.64 )
                    !***
1.00"+005-1.20"+005 !                                                                                                    0( 0.00 )
                    !
1.20"+005-1.40"+005 !***                                                                                                 1( 1.64 )
                    !***
1.40"+005-1.60"+005 !******                                                                                              2( 3.28 )
                    !******
1.60"+005-1.80"+005 !***                                                                                                 1( 1.64 )
                    !***
1.80"+005-2.00"+005 !******                                                                                              2( 3.28 )
                    !******
2.00"+005-2.20"+005 !*********                                                                                           3( 4.92 )
                    !*********
2.20"+005-2.40"+005 !***                                                                                                 1( 1.64 )
                    !***
2.40"+005-2.60"+005 !******                                                                                              2( 3.28 )
                    !******
2.60"+005-2.80"+005 !***************                                                                                     5( 8.20 )
                    !***************
2.80"+005-3.00"+005 !*********                                                                                           3( 4.92 )
                    !*********
3.00"+005-3.20"+005 !***                                                                                                 1( 1.64 )
                    !***
3.20"+005-3.40"+005 !******************                                                                                  6( 9.84 )
                    !******************
3.40"+005-3.60"+005 !******                                                                                              2( 3.28 )
                    !******
3.60"+005-3.80"+005 !***                                                                                                 1( 1.64 )
                    !***
3.80"+005-4.00"+005 !************                                                                                        4( 6.56 )
                    !************
>4.00"+005          !*********************************************************************************                  26(42.62 )
                    !*********************************************************************************
```

Fig. A3.7. Turnaround time parallel programs, first scheduling policy.

```
     HISTOGRAM PARALLEL BATCH JOBS RESPONSE TIME
MEAN VALUE= 2.08"+005
STD.DEVIATION= 6.22"+004
MIN.VALUE= 6.79"+004
MAX.VALUE= 4.46"+005
NO.OF ENTRIES=  61

<2.00"+004          *                                                                                                    0( 0.00 )
                    *
2.00"+004-4.00"+004 *                                                                                                    0( 0.00 )
                    *
4.00"+004-6.00"+004 *                                                                                                    0( 0.00 )
                    *
6.00"+004-8.00"+004 !**********************************                                                                 3( 4.92 )
                    !**********************************
8.00"+004-1.00"+005 *                                                                                                    0( 0.00 )
                    *
1.00"+005-1.20"+005 !************************                                                                           2( 3.28 )
                    !************************
1.20"+005-1.40"+005 !****************************************************************                                     6( 9.84 )
                    !****************************************************************
1.40"+005-1.60"+005 !****************************************************************************************************9(14.75 )
                    !****************************************************************************************************
1.60"+005-1.80"+005 !*************************************************                                                   5( 8.20 )
                    !*************************************************
1.80"+005-2.00"+005 !****************************************************************************************************9(14.75 )
                    !****************************************************************************************************
2.00"+005-2.20"+005 !*********************************************************************                              7(11.48 )
                    !*********************************************************************
2.20"+005-2.40"+005 !*****************************************                                                           4( 6.56 )
                    !*****************************************
2.40"+005-2.60"+005 !************                                                                                    1( 1.64 )
                    !************
2.60"+005-2.80"+005 !**********************************                                                                 3( 4.92 )
                    !**********************************
2.80"+005-3.00"+005 !**********************************                                                                 3( 4.92 )
                    !**********************************
3.00"+005-3.20"+005 !************************                                                                           2( 3.28 )
                    !************************
3.20"+005-3.40"+005 !************                                                                                    1( 1.64 )
                    !************
3.40"+005-3.60"+005 !*****************************************                                                           4( 6.56 )
                    !*****************************************
3.60"+005-3.80"+005 *                                                                                                    0( 0.00 )
                    *
3.80"+005-4.00"+005 !************                                                                                    1( 1.64 )
                    !************
>4.00"+005          !************                                                                                    1( 1.64 )
                    !************
```

Fig. A3.8. Turnaround time parallel programs, second scheduling policy

```
     HISTOGRAM PARALLEL BATCH JOBS RESPONSE TIME
MEAN VALUE= 1.49"+005
STD.DEVIATION= 3.41"+005
MIN.VALUE= 5.66"+004
MAX.VALUE= 2.51"+005
NO.OF ENTRIES=  62

                 ----------------------------------------------------------------------------------------------------
<2.00"+004       !                                                                                                    0( 0.00 )
                 !
2.00"+004-4.00"+004 !                                                                                                0( 0.00 )
                 !
4.00"+004-6.00"+004 !*******                                                                                         1( 1.61 )
                 !*******
6.00"+004-8.00"+004 !                                                                                                0( 0.00 )
                 !
8.00"+004-1.00"+005 !*****************************************                                                       6( 9.68 )
                 !*****************************************
1.00"+005-1.20"+005 !*************************************************************                                    8(12.90 )
                 !*************************************************************
1.20"+005-1.40"+005 !*************************************************************************************************** 14(22.58 )
                 !***************************************************************************************************
1.40"+005-1.60"+005 !****************************************************************                                10(16.13 )
                 !****************************************************************
1.60"+005-1.80"+005 !****************************************************************                                10(16.13 )
                 !****************************************************************
1.80"+005-2.00"+005 !*********************                                                                           3( 4.84 )
                 !*********************
2.00"+005-2.20"+005 !****************************                                                                    4( 6.45 )
                 !****************************
2.20"+005-2.40"+005 !****************************                                                                    4( 6.45 )
                 !****************************
2.40"+005-2.60"+005 !***************                                                                                 2( 3.23 )
                 !***************
2.60"+005-2.80"+005 !                                                                                                0( 0.00 )
                 !
2.80"+005-3.00"+005 !                                                                                                0( 0.00 )
                 !
3.00"+005-3.20"+005 !                                                                                                0( 0.00 )
                 !
3.20"+005-3.40"+005 !                                                                                                0( 0.00 )
                 !
3.40"+005-3.60"+005 !                                                                                                0( 0.00 )
                 !
3.60"+005-3.80"+005 !                                                                                                0( 0.00 )
                 !
3.80"+005-4.00"+005 !                                                                                                0( 0.00 )
                 !
>4.00"+005       !                                                                                                    0( 0.00 )
                 !
```

Fig. A3.9. Turnaround time parallel programs, third scheduling policy.

```
     HISTOGRAM CPU WAITING TIME FOR PARALLEL JOBS
MEAN VALUE= 3.23"+004
STD.DEVIATION= 3.93"+004
MIN.VALUE= 1.39"+003
MAX.VALUE= 9.75"+004
NO.OF ENTRIES=  61

<4.00"+003          !*****************                                                                                    2( 3.28 )
                    !*****************
4.00"+003-8.00"+003 !                                                                                                  0( 0.00 )
                    !
8.00"+003-1.20"+004 !*****************                                                                                    2( 3.28 )
                    !*****************
1.20"+004-1.60"+004 !*************************************************                                                   5( 8.20 )
                    !*************************************************
1.60"+004-2.00"+004 !*********************************************************                                         6( 9.84 )
                    !*********************************************************
2.00"+004-2.40"+004 !*****************************                                                                     3( 4.92 )
                    !*****************************
2.40"+004-2.80"+004 !*************************************************                                                   5( 8.20 )
                    !*************************************************
2.80"+004-3.20"+004 !***************************************                                                           4( 6.56 )
                    !***************************************
3.20"+004-3.60"+004 !*************************************************************************************************  9(14.75 )
                    !*************************************************************************************************
3.60"+004-4.00"+004 !*********************************************************************                             8(13.11 )
                    !*********************************************************************
4.00"+004-4.40"+004 !*************************************************************************************************  10(16.39 )
                    !*************************************************************************************************
4.40"+004-4.80"+004 !*****************************                                                                     3( 4.92 )
                    !*****************************
4.80"+004-5.20"+004 !**********                                                                                        1( 1.64 )
                    !**********
5.20"+004-5.60"+004 !**********                                                                                        1( 1.64 )
                    !**********
5.60"+004-6.00"+004 !*****************                                                                                 2( 3.28 )
                    !*****************
6.00"+004-6.40"+004 !                                                                                                  0( 0.00 )
                    !
6.40"+004-6.80"+004 !                                                                                                  0( 0.00 )
                    !
6.80"+004-7.20"+004 !                                                                                                  0( 0.00 )
                    !
7.20"+004-7.60"+004 !                                                                                                  0( 0.00 )
                    !
7.60"+004-8.00"+004 !                                                                                                  0( 0.00 )
                    !
>8.00"+004          !                                                                                                  0( 0.00 )
                    !
```

Fig. A3.10. CPU waiting time parallel programs, no multitasking, 5 PBCs.

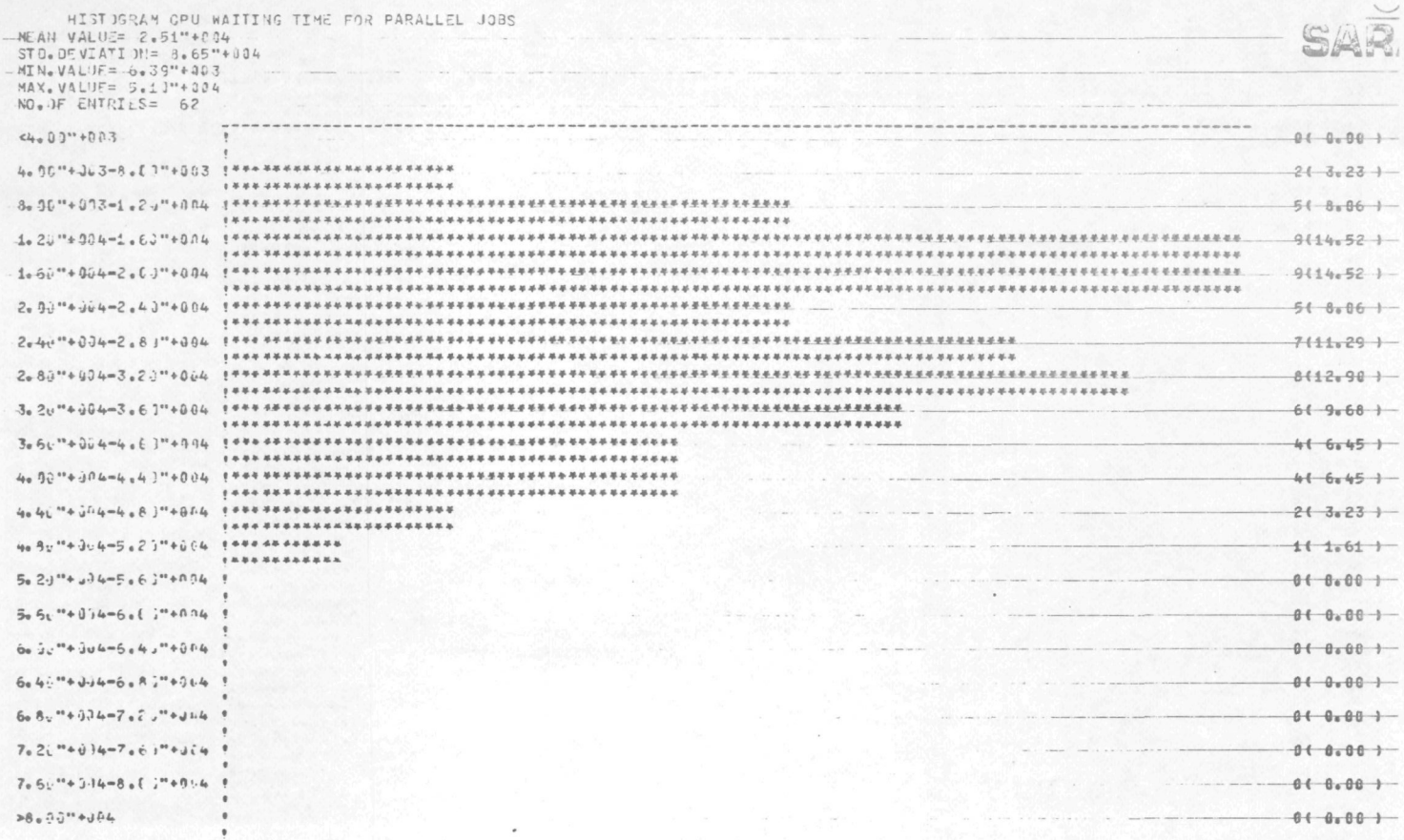


Fig. A3.11. CPU waiting time parallel programs, multitasking, 5 PBCs.

```
     HISTOGRAM CPU WAITING TIME FOR PARALLEL JOBS
MEAN VALUE= 5.57"+004
STD.DEVIATION= 2.61"+005
MIN.VALUE= 9.43"+002
MAX.VALUE= 1.20"+006
NO.OF ENTRIES=  61

<4.00"+003          ************                                                                                    2( 3.28 )
                    ************
4.00"+003-8.00"+003 ******************                                                                              3( 4.92 )
                    ******************
8.00"+003-1.20"+004 ************                                                                                    2( 3.28 )
                    ************
1.20"+004-1.60"+004 ******************                                                                              3( 4.92 )
                    ******************
1.60"+004-2.00"+004 ************                                                                                    2( 3.28 )
                    ************
2.00"+004-2.40"+004 ************                                                                                    2( 3.28 )
                    ************
2.40"+004-2.80"+004 ************                                                                                    2( 3.28 )
                    ************
2.80"+004-3.20"+004 ************                                                                                    2( 3.28 )
                    ************
3.20"+004-3.60"+004 ******************************                                                                5( 8.20 )
                    ******************************
3.60"+004-4.00"+004 ************                                                                                    2( 3.28 )
                    ************
4.00"+004-4.40"+004 *                                                                                           0( 0.00 )
                    *
4.40"+004-4.80"+004 *                                                                                           0( 0.00 )
                    *
4.80"+004-5.20"+004 *                                                                                           0( 0.00 )
                    *
5.20"+004-5.60"+004 ******************************                                                              5( 8.20 )
                    ******************************
5.60"+004-6.00"+004 ******************                                                                              3( 4.92 )
                    ******************
6.00"+004-6.40"+004 ******************                                                                              3( 4.92 )
                    ******************
6.40"+004-6.80"+004 *                                                                                           0( 0.00 )
                    *
6.80"+004-7.20"+004 ******************                                                                              3( 4.92 )
                    ******************
7.20"+004-7.60"+004 ************                                                                                    2( 3.28 )
                    ************
7.60"+004-8.00"+004 ************************                                                                        4( 6.56 )
                    ************************
>8.00"+004          ************************************************************************************************16(26.23 )
                    ************************************************************************************************
```

Fig. A3.12. CPU waiting time parallel programs, no multitasking, 10 PCBs.

```
    HISTOGRAM CPU WAITING TIME FOR PARALLEL JOBS
MEAN VALUE= 5.89"+004
STD.DEVIATION= 3.00"+005
MIN.VALUE= 3.09"+002
MAX.VALUE= 1.70"+005
NO.OF ENTRIES=  62

<4.00"+003          !*******************                                                                                    4( 6.45 )
                    !*******************
4.00"+003-8.00"+003 !**********                                                                                         2( 3.23 )
                    !**********
8.00"+003-1.20"+004 !**********                                                                                         2( 3.23 )
                    !**********
1.20"+004-1.60"+004 !*****                                                                                              1( 1.61 )
                    !*****
1.60"+004-2.00"+004 !***************                                                                                    3( 4.84 )
                    !***************
2.00"+004-2.40"+004 !*****                                                                                              1( 1.61 )
                    !*****
2.40"+004-2.80"+004 !**********                                                                                         2( 3.23 )
                    !**********
2.80"+004-3.20"+004 !*****                                                                                              1( 1.61 )
                    !*****
3.20"+004-3.60"+004 !                                                                                                   0( 0.00 )
                    !
3.60"+004-4.00"+004 !*******************                                                                                4( 6.45 )
                    !*******************
4.00"+004-4.40"+004 !************************                                                                           5( 8.06 )
                    !************************
4.40"+004-4.80"+004 !*******************                                                                                4( 6.45 )
                    !*******************
4.80"+004-5.20"+004 !***************                                                                                    3( 4.84 )
                    !***************
5.20"+004-5.60"+004 !*****                                                                                              1( 1.61 )
                    !*****
5.60"+004-6.00"+004 !*****                                                                                              1( 1.61 )
                    !*****
6.00"+004-6.40"+004 !*****                                                                                              1( 1.61 )
                    !*****
6.40"+004-6.80"+004 !*****                                                                                              1( 1.61 )
                    !*****
6.80"+004-7.20"+004 !***************                                                                                    3( 4.84 )
                    !***************
7.20"+004-7.60"+004 !*****                                                                                              1( 1.61 )
                    !*****
7.60"+004-8.00"+004 !*******************                                                                                4( 6.45 )
                    !*******************
>8.00"+004          !************************************************************************************************** 18(29.03 )
                    !**************************************************************************************************
```

Fig. A3.13. CPU waiting time parallel programs, multitasking, 10 PCBs.

SAMENVATTING

Het in dit proefschrift beschreven onderzoek is begonnen met een voorstudie naar de mogelijkheden tot opvoeren van de prestatie van computersystemen door middel van structurele verbeteringen, in plaats van het toepassen van snellere basis-schakelelementen in een conventioneel opgezet systeem. Tijdens deze voorstudie is getracht om de verscheidenheid van toepassingsmogelijkheden, waarvoor de aldus onderzochte structuren doelmatig kunnen worden toegepast, af te schatten.

Alle onderzochte structuren vertonen een zekere mate van "parallelisme" in hun werking. Het begrip "parallelisme" wordt hier gehanteerd in uitgebreide zin: het omvat "pipeline"-computers, "array"-computers, "associative"-computers, en "multi-processor"-systemen. De voorstudie leidde tot de conclusie dat "multi-processor"-systemen in het algemeen het meest flexibel zijn en dus de meeste potentiële toepassingsmogelijkheden hebben. Essentieel is daarbij wel het voorhanden zijn van een geschikt "operating system".

Naar aanleiding van de resultaten van deze voorstudie is een nader onderzoek van "multi-processor"-systemen ondernomen, waarin bijzondere aandacht is besteed aan de "operating-systems". Hierbij zijn van speciaal belang de middelen waarmee de coördinatie tussen de werkzame processen kan worden aangegeven, en meer in het algemeen het systematisch ontwerp van het "operating systeem".

Een van de beste beschikbare instrumenten voor het ontwikkelen van een "operating systeem" is de programmeertaal Concurrent PASCAL, welke in dit proefschrift is gebruikt voor het ontwerp van een model-systeem. Concurrent PASCAL maakt een veelzijdige, hiërarchisch opgebouwde organisatie van het systeem mogelijk, en verschaft doeltreffende middelen voor de coördinatie van de processen.

Het ontwerp van een "kernel" voor het "operating system" wordt beschreven, die de implementatie vormt van de basis-operaties van Concurrent PASCAL, en waar de toewijzing van de onderling gelijke verwerkingseenheden aan de hiervoor in aanmerking komende processen wordt geregeld. Bij dit ontwerp worden enkele eigenschappen die specifiek zijn voor "multi-processor operating systems" nader belicht. Deze bijzonderheden komen voort uit de

omstandigheid, dat twee (of meer) verwerkingseenheden tegelijkertijd hetzelfde gegeven trachten te verwerken, zodat maatregelen nodig zijn om dit in goede banen te leiden.

Het functionele ontwerp van een "operating system" voor een "multi-processor" wordt uitgewerkt. Het systeem is geschikt voor een combinatie van "batch"- en "time-sharing"-verwerking. Een bijzondere eigenschap van dit systeem is de mogelijkheid tot het simultaan uitvoeren, door twee of meer verwerkingseenheden, van "parallel tasks"; gedeelten van één programma die tegelijkertijd verwerkt kunnen worden, onder supervisie en coördinatie van het systeem.

Voor dit "operating system" is een simulatie-model opgesteld, en de werking ervan onder diverse omstandigheden is gesimuleerd met behulp van de programmeertaal SIMULA 67. De resultaten van deze simulatie wijzen op een aanzienlijke verbering in de totale verwerkingstijd van programma's met "parallel tasks", vergeleken met de situatie in een systeem dat de mogelijkheid van simultaan-verwerking binnen één programma niet kent.

Het proefschrift is ingedeeld in zes hoofdstukken.

- Een overzicht van de voorstudie over parallelisme in computers wordt gegeven in hoofdstuk 1.
- In hoofdstuk 2 worden de vraagstukken die verbonden zijn aan het ontwerp van "multi-processor"- "operating systems" aangegeven en toegelicht aan de hand van diverse systemen.
- Het ontwerp van de "kernel" van het model-systeem wordt behandeld in hoofdstuk 3.
- In hoofdstuk 4 vindt men het functionele ontwerp van het "operating system", beschreven als een Concurrent-PASCAL-programma.
- Het simulatie-model en een bespreking van de resultaten van de ondernomen simulatie vormen het onderwerp van hoofdstuk 5.
- Tenslotte bevat hoofdstuk 6 de uitwerking van enige conclusies die uit het onderzoek te trekken zijn, vergezeld van aanbevelingen voor mogelijke voortzetting en uitbreiding van dit onderzoek.

CURRICULUM VITAE

Born August 2, 1947 in Tapia, Romania.

Basic schooling in Tapia and Lugoj, Romania, from 1954 to 1961.

Attended the secondary school in Lugoj between 1961 and 1965.

Studied Electrical Engineering at the Polytechnic Institute of Timişoara , Romania from 1965 to 1970. Specialized in Computer Science and was granted the diploma of Electrical Engineer in 1970.

Joined the staff of the Computer Science Laboratory at the Polytechnic Institute of Timişoara in September 1970 and worked as a teaching and research assistent until November 1973. The main tasks in this period included the organization of a laboratory for peripheral devices and the assistance in several courses in programming languages.

Since December 1973 engaged in a Ph.D. project at the Delft University of Technology, The Netherlands, with a scholarship granted by the Romanian Ministry of Education.

S T E L L I N G E N

1.Multiprocessing,due to its inherent advantages,is the direction in which the development of large computer systems will evolve.

2.For a large class of applications,multitasking,as defined in this thesis,is superior to other forms of job scheduling.

3. "Reliable software" production is facilitated by a programming language which allows a hierarchical development of a system and which makes apparent and strictly controls the relations between the system components.

4.A correct description of the system workload in computer system modelling is at least as important as the correct description of the internal structure and operation of the system.But it is more difficult to define the variables that realistically characterize the workload than to describe correctly the system structure.

5.The selection of a higher level programming language to be taught to novice programmers deserves more attention than it generally receives at present.Such a language should not conceal the computers' limitations, but on the other hand it should not introduce many limitations of its own.

6.It is encouraging to see that English is almost universally accepted as the language of computer science.But it is deplorable to find that different terms are used to denote the same notion or,even worse,that the same term is used to denote different notions.

7.The social and economic implications of the rapid development of microprocessors should not be underestimated.Job requirements will be profoundly affected by bringing decentralized automation to the factory floor.The designers of systems that include microprocessors must ensure that the task of controlling such systems maintains a sufficient level of job satisfaction.

8.The excessive use of abbreviations in documents intended for a large audience impedes communication and should be made,perhaps,a punishable offence.

9.History does not repeat itself.

10.For the sake of the spiritual well-being of its citizens,a town council must pay the utmost attention to increasing the number of singing birds that live in the town.

I.Jurca

ISBN 90 6231 038 9