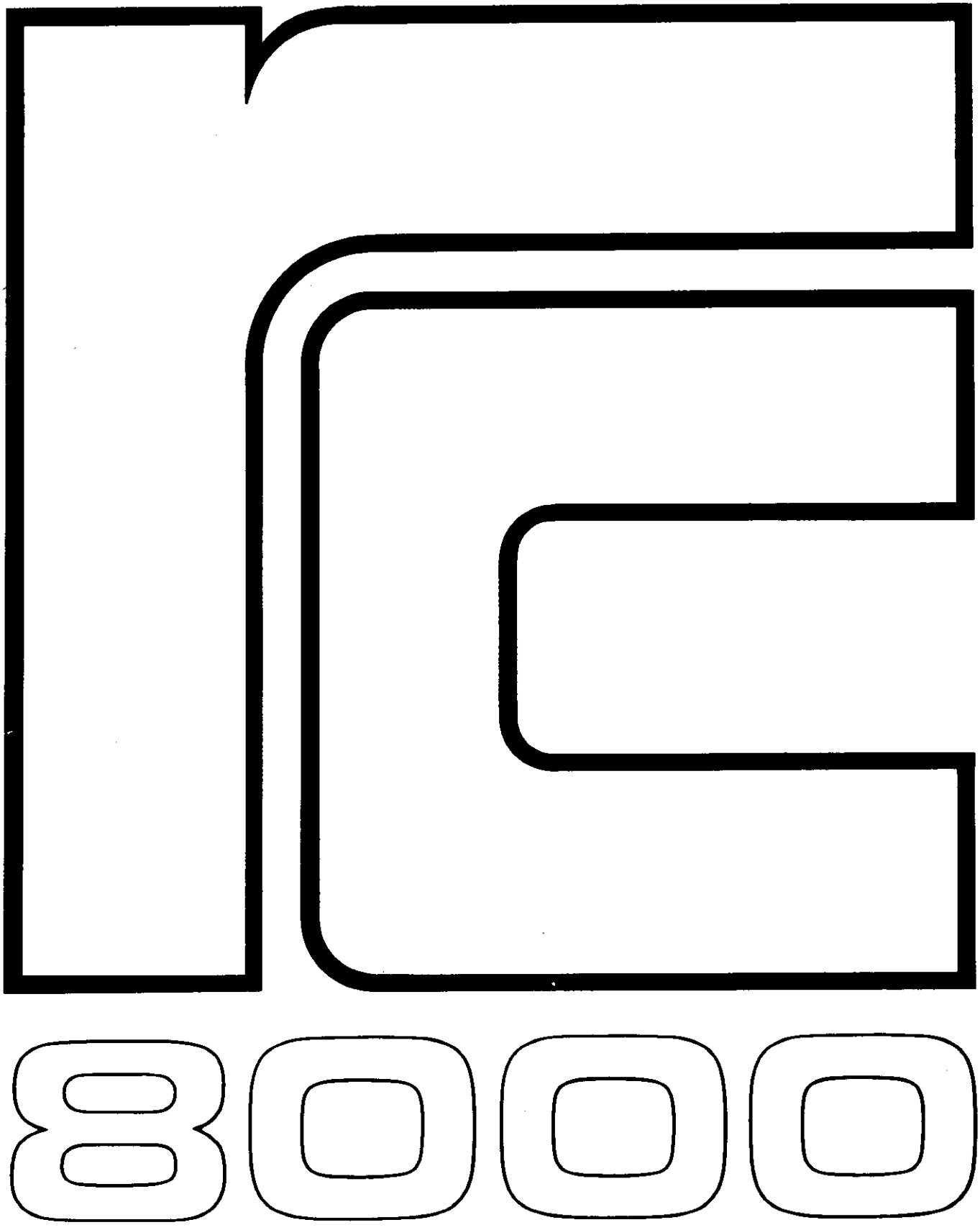
MONITOR, PART 1
System Design



RC8000 MONITOR

PART 1

SYSTEM DESIGN

Development Division
RC Computer A/S
A/S Regnecentralen af 1979

First Edition
November 1979
RCSL No. 31-D476

EDITORS: Henrik Sierslev
Pierce C. Hazelton

KEYWORDS: RC8000/6000/4000, Basic Software, Monitor, General Description

ABSTRACT: Contains a general description of the RC8000 multiprogramming system. (86 printed pages).

Printed by A/S Regnecentralen af 1979, Copenhagen

FOREWORD

The RC8000 multiprogramming system is based on the original system designed by Per Brinch Hansen, Søren Lauesen, and Peter Lindblad Andersen. It consists of a monitor program that can be extended with a hierarchy of operating systems to suit diverse requirements of program scheduling and resource allocation.

The present publication is a general description explaining the philosophy and structure of the system. It will be of interest to anyone who wishes to understand the system without going into detail about exact conventions.

The functions of the monitor and the basic operating system are further described in the publications

RC8000 Monitor, Part 2, Reference Manual

RC8000 Monitor, Part 3, External Processes

Operating System s Reference Manual

Basic programming information will be found in RC8000 Computer Family Reference Manual, and a survey of specific operating systems in An Introduction to RC8000 Operating Systems.

This publication replaces Part I of Multiprogramming System (RCSL No. 55-D140), the original monitor manual edited by Per Brinch Hansen, whom the present editors wish to thank for large portions of the text.

CONTENTS

1. SYSTEM OBJECTIVES

This chapter outlines the philosophy that guided the design of the RC8000 multiprogramming system. It emphasizes the need for different operating systems to suit different applications.

The principal goal of multiprogramming is to share a central processor and its peripheral devices among a number of programs loaded in the primary store. This is a meaningful objective if individual programs use only a fraction of the system resources and the machine is so fast, compared to the peripheral devices, that idle time within one program can be utilized by other programs.

The present system is implemented on the RC8000, a 24-bit binary computer with instruction execution times of from less than 1 microsecond to approximately 4 microseconds, depending on the model. It permits practically unlimited expansion of the primary store and standardized connection of all kinds of devices. Multiprogramming is facilitated by concurrency of program execution and input/output, program interruption, and storage protection.

The aim has been to make multiprogramming feasible on a machine with a minimum primary store of 64 K words and a fast backing store (e.g. disc). Programs can be written in any of the available programming languages. The storage protection system guarantees non-interference among parallel programs.

The system employs standard multiprogramming techniques. The central processor is shared among loaded programs. Swapping of programs in and out of the primary store is possible, but not enforced by the system. Backing storage is organized as a common data bank, in which users can store files. The system allows a conversational mode of access from consoles and terminals.

An essential part of any multiprogramming system is an operating system, i.e. a program that coordinates all computational activities and input/output. An operating system must be in complete control of the strategy of program execution, and assist users with such functions as operator communication, interpretation of job control statements, allocation of resources, and application of execution time limits.

For the designer of advanced information systems, it is essential that the operating system will allow him to change the mode of operation which it controls; if this is not the case, his freedom of design may be seriously restricted. Unfortunately this is precisely what many operating systems do not allow, being based exclusively on a single mode of operation, e.g. batch processing, priority scheduling, real-time scheduling, or time-sharing.

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

This state of affairs would indicate that the main problem in the design of a multiprogramming system is not to define functions that satisfy specific operating needs, but rather to provide a system nucleus that can be extended with new operating systems in an orderly manner. This is the prime objective of the RC8000 multiprogramming system.

The nucleus of the RC8000 multiprogramming system is a monitor program with complete control of storage protection, input/output, and interrupts. Essentially the monitor is a software extension of the hardware structure which makes the RC8000 attractive for multiprogramming. The following elementary functions are implemented in the monitor:

- Scheduling of time slices among programs executed in parallel by means of a digital clock.

- Initiation and control of program execution at the request of other running programs.

- Transfer of messages between running programs.

- Supervision of data transfers to or from peripheral devices.

The monitor has no built-in strategy of program execution and resource allocation, but allows any program to initiate other programs in a hierarchical manner and to execute them according to any strategy desired. In this hierarchy of programs an operating system is simply a program that controls the execution of other programs.

Thus operating systems can be introduced into the system, just as other programs, without modification of the monitor. Furthermore, operating systems can be replaced dynamically, enabling each installation to switch between various modes of operation; several operating systems can, in fact, be active simultaneously.

This dynamic operating system concept will be explained in detail in the following chapters. Particular strategies of program scheduling, however, will not be described, which is in keeping with the system design philosophy, and the discussion will concentrate on the fundamental aspects of the control of an environment of parallel processes.

2. ELEMENTARY MULTIPROGRAMMING PROBLEMS 2.

This chapter introduces the elementary multiprogramming problems of mutual exclusion and mutual synchronization of parallel processes. The discussion is confined to the logical problems that arise when independent processes attempt to access common variables and shared resources. An understanding of these concepts is indispensable for the reader who wishes to appreciate the difficulties of changing from monoprogramming to multiprogramming.

2.1 Multiprogramming 2.1

In multiprogramming the sharing of computing time among programs is controlled by interrupts, which activate a monitor program. The monitor saves the state of the interrupted program and allocates the next slice of computing time to another program, and so on. Switching from one program to another is also performed whenever a program must wait for the completion of input/output.

Thus although the computer can execute only one instruction at a time, multiprogramming creates the illusion that programs are being executed simultaneously.

2.2 Parallel Processes 2.2

Most of the elementary problems in multiprogramming arise from the fact that one process, e.g. an executing program, can make no assumptions about the relative speed and progress of other processes. This is a potential source of conflict whenever two processes attempt to access a common variable or a shared resource.

This problem will obviously exist in a truly parallel system, in which programs are executed simultaneously on several processors, but one should realize that it will also exist in a quasi-parallel system based on the sharing of a single processor by means of interrupts: Since a program cannot detect when it has been interrupted, it does not know how far other programs have progressed.

To put it another way, if we consider the system as seen from within a program, it is immaterial whether multiprogramming is implemented on one or more processors - the logical problems are the same.

Consequently a multiprogramming system must in general be viewed as an environment containing a number of truly parallel processes. Having reached this conclusion, a natural generalization is to treat not only program execution but input/output also as independent, parallel processes. This point will be amply illustrated in the following chapters.

2.3 Mutual Exclusion

The idea of multiprogramming is to share the computing equipment among several parallel programs. At any one moment, however, a given resource may belong to one program only. In order to ensure this, we must introduce global variables which programs can inspect to determine whether a given resource is available.

As an example consider a console used by all programs for messages to the operator. To control access to this device we might introduce a global variable console available. When a program p wishes to display a message, it must examine and set this variable by means of the following instructions:

```
wait:  load      console available
       skip if   true
       jump to   wait
       load      false
       store     console available
```

Assume now that program p is interrupted after it has loaded the variable, but before it has been able to examine and set it. The register containing the value of the variable is stored within the monitor, and program q, let us say, is started. Program q loads the same variable and finds that the console is available; it therefore assigns the value false to the variable and starts using the console. Soon q is interrupted, and later p is restarted with the original contents of the register re-established by the monitor. Program p now inspects the original value of the variable and concludes, erroneously, that the console is available.

This conflict arises because programs have no control over the interruption system; thus the only indivisible operations available to programs are single instructions, e.g. load, compare, or store. This example shows that one cannot implement a multiprogramming system without ensuring a mutual exclusion of programs during the inspection of global variables. It is evident that the entire reservation sequence must be executed as an indivisible function. One of the purposes of a monitor program is to execute indivisible functions in the interrupt disabled mode.

In using reservation primitives, one must be aware of the problem of the deadly embrace between two processes, p and q, which attempt to share the resources r and s as follows:

```
p:  wait and reserve(r)---wait and reserve(s)-
q:  wait and reserve(s)---wait and reserve(r)-
```

This can cause both processes to wait forever, since neither realizes that it wants what the other has.

To avoid this problem we require a third process, an operating system, which controls the allocation of shared resources between p and q in a manner that guarantees that both will be able to proceed to completion (if necessary by delaying the one until resources become available).

2.4 Mutual Synchronization 2.4

In a multiprogramming system, parallel processes must be able to cooperate in the sense that they can activate one another and exchange information. One example of a process activating another process is the initiation of input/output by a program. Another example is that of an operating system that schedules a number of programs. The exchange of information between two processes can be regarded as a problem of mutual exclusion in which the receiver must be prevented from inspecting the information until the sender has delivered it in a common storage area.

Since the two processes are independent with respect to speed, it is not certain that the receiver is ready to accept the information when the sender chooses to deliver it; conversely, the receiver can become idle when there is no more information for it to process.

This problem of the synchronization of two processes during a transfer of information must be solved by indivisible monitor functions which allow a process to be delayed at its own request and activated at the request of another process.

3. BASIC MONITOR CONCEPTS

3.

This chapter opens a detailed description of the RC8000 monitor. A multiprogramming system is viewed as an environment in which program execution and input/output are handled uniformly as cooperating, parallel processes. The need for an exact definition of the process concept is stressed. The purpose of the monitor is to bridge the gap between the actual hardware and the abstract concept of multiprogramming.

3.1 Introduction

3.1

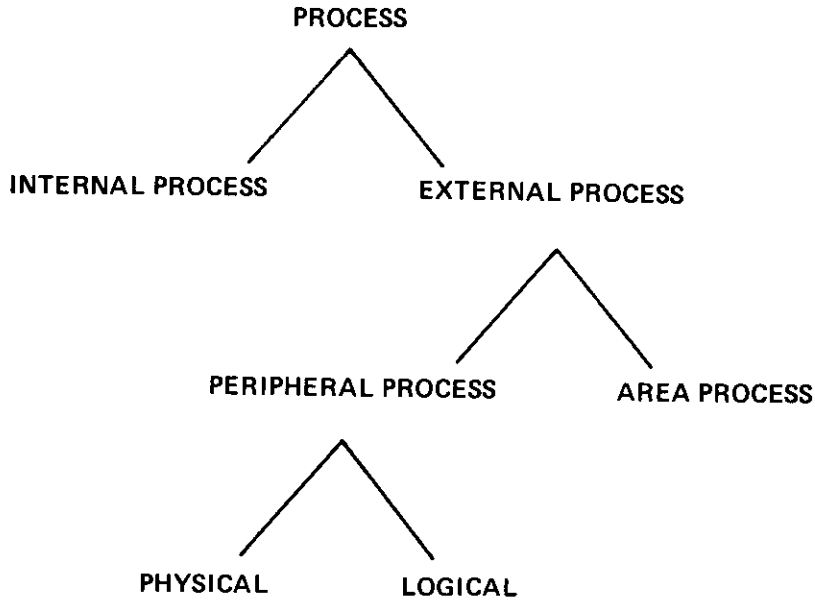
The aim has been to implement a multiprogramming system which can be extended with new operating systems in a well-defined manner. In order to do this, a sharp distinction must be made between the control and the strategy of program execution.

The mechanisms provided by the monitor solve the logical problems of the control of parallel processes. They also solve the security problems that arise when erroneous or malicious processes attempt to interfere with other processes. They leave, however, the choice of particular strategies of program scheduling to the processes themselves, i.e. to the operating systems.

In order to realize this objective, the following basic mechanisms have been implemented within the monitor:

- Simulation of parallel processes.
- Communication between processes.
- Creation, control, and removal of processes.

Let us now assign a precise meaning to the process concept, i.e. introduce an unambiguous terminology for what a process is and how it is implemented on the RC8000 computer.

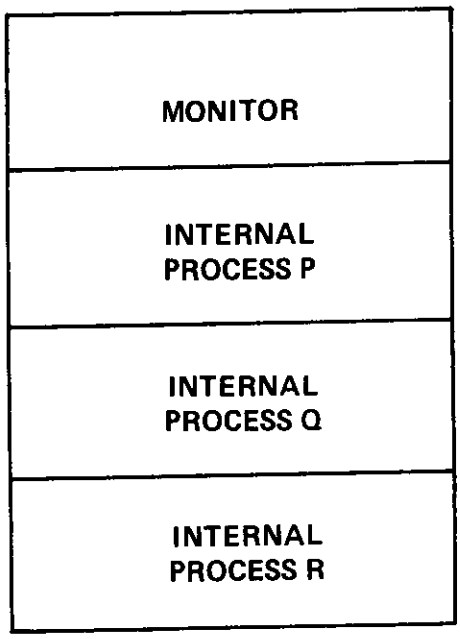


As may be seen from the figure above, a fundamental distinction is made between internal processes and external processes; the former correspond roughly to program execution, the latter to input/output.

3.2 Programs and Internal Processes

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

The following figure illustrates the division of the primary store between the monitor and three internal processes, p, q, and r.



Later, in Chapter 6, we shall explain how internal processes are created and how programs are loaded into them. Here it is sufficient to point out that an internal process occupies a contiguous storage area of a fixed size during its entire lifetime.

The monitor maintains a process description for each internal process. This table contains such information as the name, storage area, and current state of the process.

Computing time is shared cyclically among all active internal processes. The monitor allocates a maximum time slice of 25.6 milliseconds to each internal process in turn; when an interrupt occurs, the process is interrupted and its state is stored in the process description; the monitor then allocates 25.6 milliseconds to another internal process, and so on. The cyclic queue of active internal processes is called the time-slice queue.

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

An internal process like p may involve the execution of a sequence of programs, e.g. editing followed by translation and execution of an object program. It is also possible for copies of the same program, say, a compiler, to be executed simultaneously in two processes, q and r. These examples show the need for a distinction between programs and processes.

3.3 Documents and External Processes 3.3

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

A peripheral device is an item of equipment connected to a device controller and identified by a device number.

A document is a collection of data stored on a physical medium, e.g. a roll of paper tape, deck of punched cards, printer form, reel of magnetic tape, or file on a backing-storage device.

By the expression external process we refer to the input/output of a given document identified by a unique process name. This concept implies that once a document has been mounted, internal processes can refer to it by name without knowing the actual device employed.

The monitor maintains a process description for each external process. This table contains such information as the name, kind, and current state of the process. It also contains two bit masks: the one indicates potential users, i.e. which internal processes may read the document mounted on the device; the other indicates reservation by the current user, i.e. whether one of these internal processes may write on the document.

The use of external processes is explained in detail in Chapter 5.

3.3.1

3.3.1 Peripheral Processes

An external process which describes a peripheral device is called a peripheral process. The process description for a peripheral process will therefore contain the number of the device represented, in addition to the information mentioned in Section 3.3.

Some devices, e.g. discs, can be divided into a number of logical devices. In these cases there will be a peripheral process description (containing the device number) for the physical device and a peripheral process description (containing a reference to the process description with the device number) for each logical device.

3.3.2

3.3.2 Area Processes

An external process which describes a file on a backing-storage device is called an area process. Area processes are further explained in Chapter 8.

3.4 Pseudo Processes 3.4

A pseudo process is a description of an internal or external process which permits the latter to appear under a pseudonym. All communications sent to a pseudo process are forwarded to the internal process that created it.

Let us assume that an operating system creates a pseudo process named printer. The pseudo process will then intercept all communications (from internal processes running under the operating system) to the external process named printer and forward them to the operating system.

In this way the operating system can determine when and where output printing for the jobs running under it will take place.

3.5 Bases and the Protection System 3.5

In a multiuser environment the monitor must ensure that processes do not interfere with one another. The protection system therefore encompasses:

1. Storage protection, which is implemented by means of two fields in the internal process description. These fields define the primary storage area assigned to the internal process (see further Sect. 7.3).

2. File protection, which is implemented by assigning three so-called bases to the internal process. These bases, each of which is represented by two integers, define which backing-storage files the internal process may read and write in (see further Sect. 6.1 and 8.3.4).

3.6 Monitor

The monitor is not considered an independent process, but rather a software extension of the hardware structure which makes the RC8000 computer attractive for multiprogramming. The functions of the monitor are these:

1. To maintain descriptions of all processes.
2. To share computing time among internal processes.
3. To implement procedures which processes can call in order to create and control other processes and communicate with them.

The monitor is a program which is activated by interrupts. It can execute privileged instructions and run in the interrupt disabled mode. This implies:

1. That the monitor is in complete control of input/output, storage protection, and the interruption system.
2. That the monitor can execute a sequence of instructions as an indivisible entity.

After initial system loading the monitor resides permanently in the primary store (in its lowest part).

3.7 Summary

The multiprogramming system has thus far been described as a set of independent, parallel processes identified by names. The emphasis has been on a clear understanding of relationships between:

- Resources (storage and peripheral devices).
- Data (programs and documents).
- Processes (internal and external).

4. PROCESS COMMUNICATION

4.

This chapter deals with the monitor procedures for the exchange of information between two parallel processes. The mechanism of message buffering is defended on the grounds of security and efficiency.

4.1 Message Buffers and Queues

4.1

Two parallel processes can cooperate by sending messages to each other.

A message consists of eight words. Messages are transmitted from one process to another by means of message buffers selected from a common pool within the monitor.

The monitor administers a message queue for each process. Messages are linked to this queue when they arrive from other processes. The message queue is part of the process description.

Normally a process serves its queue on a first-come, first-served basis. After the processing of a message, the receiving process returns an answer of eight words to the sending process in the same buffer.

As described in Section 2.4, communication between two independent processes requires a synchronization of the processes during a transfer of information. A process requests synchronization by executing a wait operation; this causes a delay of the process until another process executes a send operation.

The term delay means that the internal process is removed temporarily from the time-slice queue; the process is said to be activated when it is again linked to the time-slice queue.

4.2 Send and Wait Procedures

The following monitor procedures are available for communication between internal processes:

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

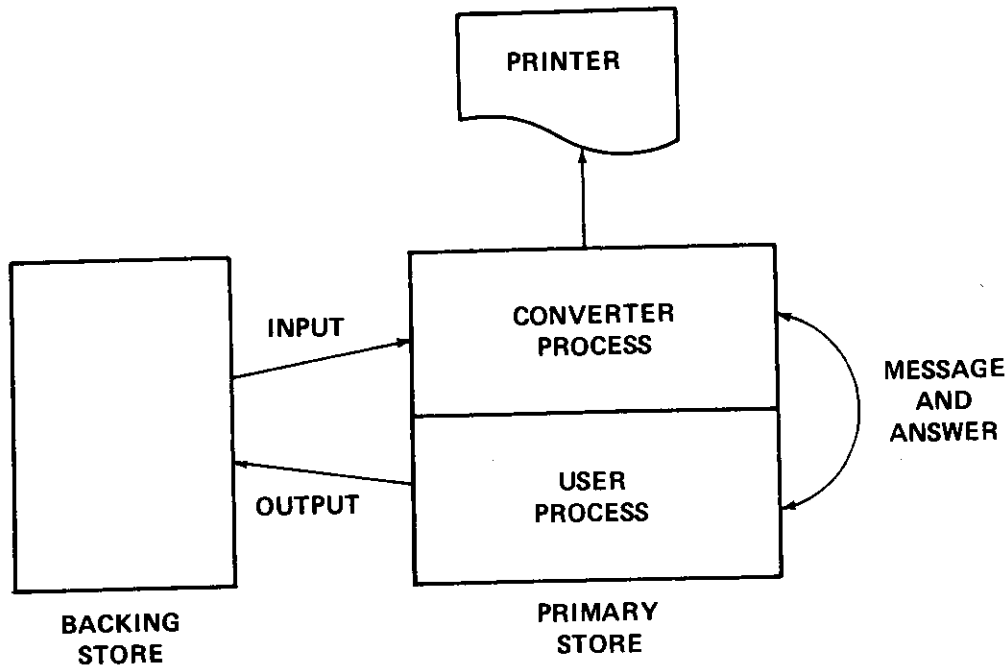
Send message copies a message into the first available buffer within the pool and delivers it in the queue of a named receiver. The receiver is activated if it is waiting for a message. The sender continues after being informed of the address of the message buffer.

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

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

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

The use of these procedures may be illustrated by the following example of a conversational process. The figure below shows one of several user processes which deliver their output on the backing store. After the completion of its output a user process sends a message to a converter process requesting it to print the output. The converter process receives and serves these requests one by one, thus ensuring that the printer is shared by all user processes with a minimum of delay.



The algorithms of the converter process and the user process are as follows:

```
converter:  wait message(sender,message,buffer);
            print from backing store(message);
            send answer(result,answer,buffer);
            goto converter;

user:   ---
        output on backing store;
        send message(<:converter:>,message,buffer);
        wait answer(result,answer,buffer);
        ---
```

4.3 General Event Procedures

The communication procedures enable a conversational process to receive messages simultaneously from several other processes. To avoid becoming a system bottleneck, however, a conversational process must be prepared to engage actively in more than one conversation at a time.

As an example, think of a conversational process that engages itself, at the request of another process, in a conversation with one of several human operators in order to have some manual operation performed (e.g. the mounting of a tape).

If one restricts a conversational process to accepting only one request (message) at a time, and to completing the requested action before receiving the next request, the unacceptable consequence is that other processes (including human operators at consoles) may have their requests delayed for long or even undefined periods of time.

As soon as a conversational process has initiated a lengthy action, by sending a message to some other process, it must receive further messages and initiate other actions. It will then be reminded later on of the completion of earlier actions by means of normal answers. In general a conversational process is now engaged in several requests at one time.

This introduces a scheduling and resource problem: When the process receives a request, some of its resources (storage or peripheral devices) may be occupied by already initiated actions; thus in some cases the process will not be able to honor new requests until old ones are completed.

In such a situation the process would like to postpone the reception of some requests and leave them pending in the queue, while examining others.

The procedures wait message and wait answer, which force a process to serve its queue in a strict sequential order and delay itself while its own requests to other processes are completed, do not fulfill the above requirements.

Two more general communication procedures have therefore been introduced, which enable a process to wait for the arrival of the next message or answer and serve its queue in any order:

```
wait event(last buffer,next buffer,result)
get event(buffer)
```

The term event denotes a message or answer. The queue of a process will accordingly be called the event queue from now on.

Wait event delays the calling process until either a message or an answer arrives in its queue after a given last buffer. The process is supplied with the address of the next buffer and a result indicating whether it contains a message or an answer. If the address of the last buffer is zero, the queue is examined from the beginning. The procedure does not remove the next buffer from the queue nor change its status in any other way.

As an example, consider an event queue with two pending buffers, a and b:

queue = buffer a, buffer b

The calls wait event(0,buffer) and wait event(a,buffer) will cause immediate return to the process with buffer equal to a and b, respectively, whereas the call wait event(b,buffer) will delay the process until another message or answer arrives in the queue after buffer b.

Get event removes a given buffer from the queue of the calling process. If the buffer contains a message, it is made ready for the sending of an answer; if it contains an answer, it is returned to the common pool. The copying of the message or answer from the buffer must be done by the process itself before get event is called (see the RCSL publication RC8000 Monitor, Part 2: Reference Manual).

The following algorithm illustrates the use of these procedures within a conversational process:

```
first event:
   buffer:=0;
next event:
   last buffer:=buffer;
   wait event(last buffer,buffer,result);
   if result=message then
   begin
examine request:
   if resources not available then goto next event;
initiate action:
   get event(buffer);
   reserve resources;
   ---
   send message to some other process;
   save state of action;
   end else
   begin comment: result=answer;
terminate action:
   restore state of action;
   get event(buffer);
   release resources;
   send answer to original sender;
   end;
   goto first event;
```

The process starts by examining its queue; if the queue is empty, it awaits the arrival of the next event. If it finds a message, it checks whether it has the necessary resources to perform the requested action; if not, it leaves the message in the queue and examines the next event; otherwise it accepts the message, reserves the resources, and initiates the action.

As soon as this involves the sending of a message to some other process, the conversational process saves the state of the incompleted action and proceeds to examine its queue from the beginning in order to engage itself in another action.

Whenever the process finds an answer in its queue, it immediately accepts it and completes the corresponding action. It can now release the resources used and send an answer to the original sender that made the request. After this, it examines the entire queue again to see whether the release of resources has made it possible to accept pending messages.

One example of a process operating in accordance with this scheme is the basic operating system, s, which creates internal processes at the request of console operators. s can engage in conversations with several consoles at the same time. It will postpone a request only if its storage is occupied by other requests or if it is already in the middle of an action requested from the same console (see the RCSL publication Operating System s Reference Manual).

4.4 Advantages of Message Buffering

4.4

In the design of the communication scheme, full recognition has been given to the fact that the multiprogramming system is a dynamic environment, in which some processes may turn out to be black sheep.

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

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

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

Efficiency is obtained by the queuing of buffers, which enables a sending process to continue immediately after the delivery of a message or answer regardless of whether the receiver is ready to process it.

In order for the system to be dynamic, it is vital that a process can be removed at any time, even if it is engaged in one or more conversations. In the previous example, of user processes that deliver their output on the backing store and ask a converter process to print it, it would be reasonable to remove a user process which had completed its task and was now waiting only for an answer from the converter process. When this is the case, the monitor leaves all messages from a removed process undisturbed in the queues of other processes. When these processes terminate their actions by sending answers, the monitor simply returns the buffers to the common pool.

The reverse is also possible: During the removal of a process, the monitor may find unanswered messages sent to the process. These are returned as dummy answers to the senders. A special instance of this is the generation of a dummy answer to a message addressed to a non-existent process.

The main drawback of message buffering is that it introduces yet another resource problem, as the common pool contains a finite number of buffers. If a process were allowed to empty the pool by sending messages to ignorant processes which did not respond with answers, further communication within the system would be blocked. A limit has therefore been placed on the number of messages which a process can send simultaneously. This, and the fact that a process is allowed to transmit an answer in a received buffer, places the entire risk of a conversation on the process that opens it (see Sect. 7.4).

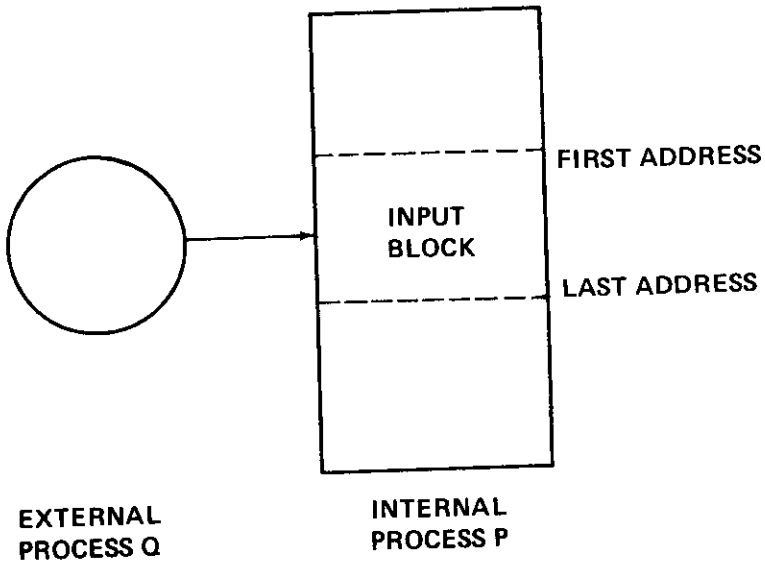
5. EXTERNAL PROCESSES

This chapter clarifies the meaning of the external process concept. It explains the initiation of input/output by means of messages from internal processes, exclusive access to documents by means of reservation, and the dynamic creation and removal of external processes. It concludes with a brief discussion of links.

5.1 Initiation of Input/Output

When an internal process wishes to use a peripheral device, it sends a message to the relevant external process.

Consider the following situation, in which an internal process, p, inputs a block of data from an external process, q, say, a magnetic tape:



Process p initiates input by sending a message to q:

```
send message(<:q:>,message,buffer)
```

The message consists of eight words defining an input/output operation and the first and last addresses of a storage area within process p:

```
message:  operation
          first storage address
          last storage address
          (five irrelevant words)
```

The monitor copies the message into a buffer and delivers it in the queue of process q; it then uses the kind parameter in the process description of q to switch to a piece of code common to all magnetic tape stations. If the tape station is busy, the message is simply left in the queue; otherwise input is initiated to the given storage area. On return, program execution continues in process p.

When the tape station completes input by means of an interrupt, the monitor generates an answer and, using the process description of q once more, delivers it in the queue of p. Process p, in turn, accepts the answer by calling

```
wait answer(result,answer,buffer)
```

The answer contains the status bits sensed from the device and the actual block length expressed as the number of 12-bit halfwords and the number of characters input:

```
answer:  status bits
         number of halfwords
         number of characters
         (five irrelevant words)
```

After having delivered the answer, the monitor examines the queue of process q and initiates the next operation (unless the queue is empty).

All external processes essentially follow this scheme, which can be defined by the algorithm

```
external process:  wait message;
                   analyze and check message;
                   initiate input/output;
                   wait interrupt;
                   generate answer;
                   send answer;
                   goto external process;
```

5.2 Reservation and Release

The use of message buffering provides a direct means of sharing an external process among a number of internal processes: An external process can simply accept messages from any internal process and serve them in the order of their arrival. An example of this is the use of a single console for the display of messages to an operator.

This method of sharing a device ensures that a block of data is input or output as an indivisible entity. When a sequential medium (e.g. paper tape, punched cards, or magnetic tape) is used, however, an internal process must have exclusive access to the entire document.

Exclusive access is obtained by means of the monitor procedure

reserve process(name,result)

The result indicates whether the reservation has been accepted.

An external process that handles sequential documents of this kind rejects messages from all internal processes save the one that has reserved it. Rejection is indicated by the result of the procedure wait answer.

During the removal of an internal process, the monitor removes all reservations made by it. Internal processes can also do this explicitly, however, by means of the monitor procedure

release process(name)

5.3 Creation and Removal

From the operator's point of view, an external process is created when he mounts a document on a device and names it. The name, however, must be communicated to the monitor by means of an operating system, i.e. an internal process that controls the execution of programs. Thus it is more accurate to say that external processes are created when internal processes assign names to peripheral devices. This is done by means of the monitor procedure

create peripheral process(name,device number,result)

The monitor does not, in fact, always have a means of ensuring that a given document is mounted on a device. Furthermore, certain devices operate without documents, e.g. the real-time clock (see Sect. 8.2).

The name of an external process can be explicitly removed by a call of the monitor procedure

```
remove process(name,result)
```

The monitor automatically removes the process name when it detects operator intervention in the operation of a device.

5.4 Links

Peripheral devices are connected to a device controller. The controller for centrally connected devices is placed in a front-end device controller, which is physically integrated in the RC8000. The controller for remote devices is placed in a concentrator, which is connected to the front-end device controller by data transmission lines.

The logical data path between the peripheral process description and the physical device is called a link. The link concept implies that a user process can access any device without being aware of its geographical location. The link, moreover, permits the user himself to select any computer in the network as his job host and, within it, any application.

A link is either permanent or temporary, as specified at the time of its creation. Links can be created in the following ways:

1. Implicitly by the monitor (on initial system loading) when specified as monitor options. Such links are always permanent.
2. Explicitly by an operating system or other user program.
3. From a terminal, either implicitly or by command.

The creation of a link will always cause the creation of a peripheral process. The device may then be renamed by means of the procedure create peripheral process.

Links are removed only by request. If the transmission line is disconnected, however, a temporary link will be removed automatically, and must be re-created when the line is connected again.

Links are more fully described in the RCSL publication RCNET, General Information.

6. INTERNAL PROCESSES

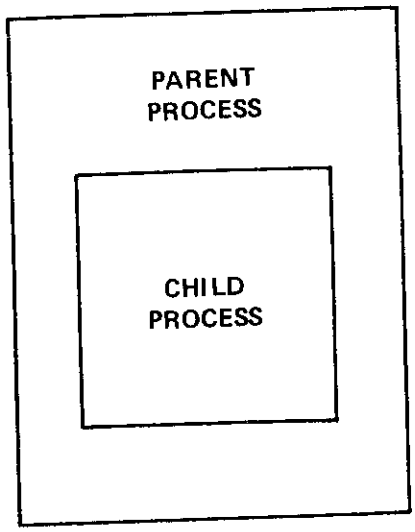
This chapter explains the creation and control of internal processes. The emphasis is on the hierarchical structuring of internal processes, which makes it possible to extend the system with new operating systems. The dynamic behavior of the system is explained in terms of process states and the transition from one state to another.

6.1 Creation, Control, and Removal

Internal processes are created at the request of other internal processes by means of the monitor procedure

```
create internal process(name,parameters,result)
```

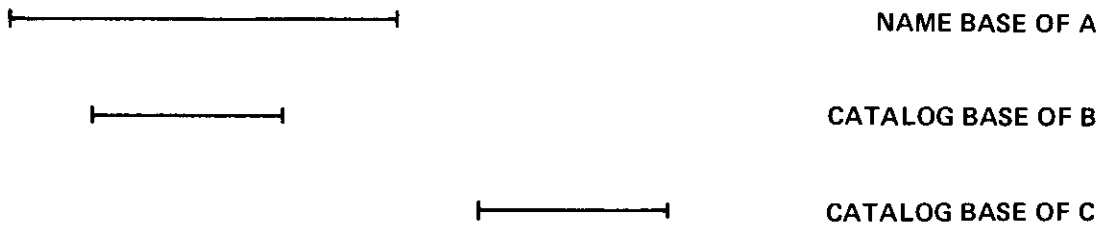
The monitor initializes the process description of the new process with the process name and storage area assigned by the parent process. The storage area of the child process must be within its parent's area:



When an internal process is created, it is assigned four bases, each of which is represented by two integers.

The parent process defines the max base and the standard base of the child process as being inside or equal to its own max base and standard base, and the monitor sets the catalog base of the child equal to the standard base defined by the parent. The max, standard, and catalog bases determine, among other things, which files the internal process may read and write in (see further Sect. 8.3.4).

The name assigned to the process has two parts: the name proper (up to 11 ISO characters) and a name base. The monitor sets the name base of the child process equal to the catalog base of the parent process. The name base determines which internal processes can see the new process, viz. all those whose catalog bases are inside or equal to the name base of the new process. In the following figure, process b can see a, but process c cannot. (Note that we cannot conclude from this whether a can see b or c).



After its creation the child process is simply a named storage area, which is described within the monitor. It has not yet been linked to the time-slice queue.

The parent process can now load a program into the child process by means of an input operation. Following this, the parent can initialize the registers of its child using the monitor procedure

```
modify internal process(name,registers,result)
```

The register values are stored in the process description until the child process is started. As a standard convention adopted by parent processes (but not enforced by the monitor), the registers inform the child about the process descriptions of itself, its parent, and the console which it may use for operator communication.

Finally the parent can start program execution within the child by calling

```
start internal process(name,result)
```

which links the child to the time-slice queue. The child now shares time slices with other active processes including its parent.

At the request of a parent process, the monitor will wait for the completion of all input/output initiated by a child process and then stop it, i.e. remove it from the time-slice queue:

```
stop internal process(name,buffer,result)
```

The significance of the message buffer will be explained in Section 6.3.

In the stopped state, a child process can be modified and started again, or it can be completely removed by the parent process:

remove process(name,result)

which causes all resources borrowed by the child to be returned to the parent. During removal the monitor generates dummy answers to all messages sent to the child process and releases all external processes utilized by it. The parent process can now use the storage area to create other child processes.

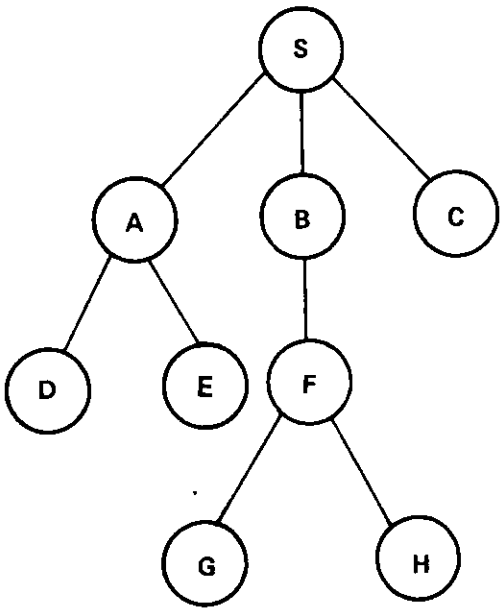
6.2 Process Hierarchy 6.2

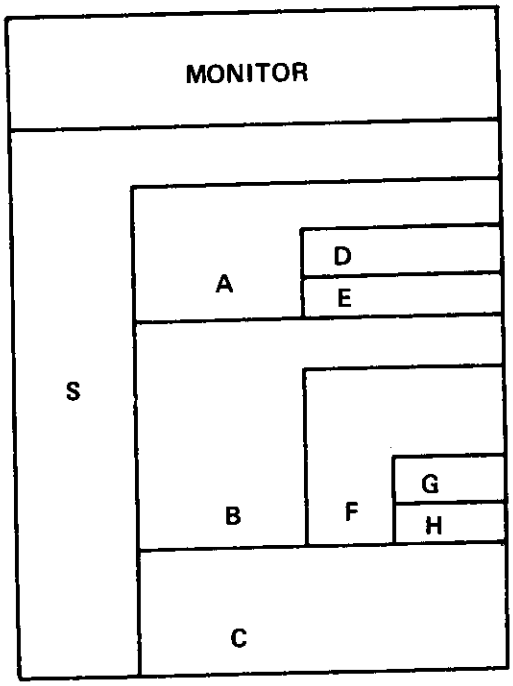
The idea of the monitor has been described as the simulation of an environment in which program execution and input/output are handled uniformly as parallel, cooperating processes. A basic set of procedures permits the dynamic creation and control of processes as well as communication between them.

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

After initial system loading, the primary store contains the monitor and an internal process, called s, which is the basic operating system. s can create parallel processes, a, b, c, and so on, at the request of console operators and terminal users. These processes in turn can create other processes, d, e, f, and so on.

Thus while s acts as a primitive operating system for a, b, and c, the latter in turn act as operating systems for their children, d, e, f, and so on. This is illustrated by the following figure, which shows a family tree of processes on the left and the corresponding storage allocation on the right:





This family tree of processes can be extended to any level, subject only to a limitation on the total number of processes. At present the maximum number of internal processes which can be created under the basic operating system is 21.

In this multiprogramming system all privileged functions are implemented in the monitor, which has no built-in strategy. Strategies can be introduced on the various higher levels, where each process has the power to control the scheduling and resource allocation of its own children.

The only rules enforced by the monitor are these:

1. A process can only allocate a subset of its own resources, including storage, to its children.
2. A process can only modify, start, stop, and remove its own children.

The structure of the family tree is defined in the process descriptions within the monitor. We emphasize that the only function of the tree is to define the basic rules of process control and resource allocation. Time slices are shared evenly among active processes, regardless of their position in the hierarchy, and each process can communicate with all other processes.

With regard to the development of operating systems, the most important characteristics can now be seen as the following:

1. New operating systems can be implemented as other programs without modification of the monitor. In this connection we should mention that the ALGOL language for the RC8000 contains facilities for calling the monitor and initiating parallel processes. Thus it is possible to write operating systems in a high-level language.

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

3. Utility programs and user programs can be executed under different operating systems without modification; this is ensured by a standardization of communication between parents and children.

6.3 Process States

We are now able to define the possible states of an internal process as described within the monitor. An understanding of the transition from state to state is vital as a key to the dynamic behavior of the system.

An internal process is either running (executing instructions or ready to do so) or waiting (for an event outside the process). In the running state the process is linked to the time-slice queue; in the waiting state it is temporarily removed from this queue. A process can be waiting for a message, an answer, or an event, as explained in Chapter 4.

More complex are those states in which a process is waiting to be stopped or started by another process. In order to explain this, we shall refer once more to the family tree shown in the previous section.

Let us say that process b wants to stop its child, f. The purpose in doing this is to ensure that all program execution and input/output within the storage area of f are stopped. Since part of this storage area has been allocated to children of f, it is obviously necessary to stop not only the child f but also all descendants of f. This is complicated by the fact that some of these descendants may have already been stopped by their own parent. Thus in the present example, process g may still be running, while process h may have been stopped by the parent, f. Consequently the monitor should stop only f and g.

Consider now the reverse situation, in which process b starts its child, f, again. Now the purpose is to re-establish the process states that prevailed before f was stopped.

Here the monitor must be careful to start only those descendants of f which were stopped along with f. Thus in our example, the monitor must start f and g, but not h; otherwise we confuse f, which still assumes that its child h is stopped.

Clearly, then, the monitor must distinguish between processes that are stopped by their parents and processes that are stopped by their ancestors.

The possible states of an internal process are these:

running
running after error
waiting for message
waiting for answer
waiting for event
waiting for start by parent
waiting for stop by parent
waiting for start by ancestor
waiting for stop by ancestor
waiting for process function

A process is created in the state waiting for start by parent. When it is started, its state becomes running. The state running after error is explained in Section 8.1.

When a parent wishes to stop a child, the state of the child is changed to waiting for stop by parent, and all running descendants of the child are described as waiting for stop by ancestor. At the same time these processes are removed from the time-slice queue.

What remains to be done is to ensure that all input/output initiated by these processes is terminated. In order to control this, each internal process description contains an integer called the stop count.

The stop count is increased by one each time the internal process initiates input/output from an external process. On the completion of an operation, the monitor decreases the stop count by one and examines the state of the internal process. If the stop count becomes zero and the process is waiting for stop by parent (or ancestor), its state is changed to waiting for start by parent (or ancestor).

Only when all processes involved are waiting for start is the stop operation completed. This may take time, and it may not be acceptable to the parent (an operating system with many other tasks) to remain inactive for so long. For this reason the stop operation is divided into two parts. The stop procedure

```
stop internal process(name,buffer,result)
```

only initializes the stopping of a child and selects a message buffer for the parent. When the child and its running descendants are completely stopped, the monitor delivers an answer to the parent in this buffer. Thus the parent can use the procedure wait answer or wait event to wait for the completion of the stop.

A process can be in any state when a stop is initiated. If it is waiting for a message, answer, or event, its state will be changed to waiting for stop, as explained above, but at the same time its instruction counter will be decreased by two so that it can repeat the call of wait message, wait answer, or wait event when it is started again.

It should be noted that messages and answers can be delivered in the queue of a process in any state. This ensures that a process does not lose touch with its surroundings while it is stopped.

The state waiting for process function is explained in the next section.

6.4 Interruptable Monitor Functions 6.4

Certain monitor functions are executed by an auxiliary internal process. These so-called process functions are generally speaking all those which involve changes in the hierarchy of names and those which are too time-consuming to be executed entirely in the interrupt disabled mode. The auxiliary process runs only for brief intervals in the interrupt disabled mode; otherwise it shares computing time on an equal footing with other internal processes.

When an internal process calls a process function, the following takes place: The calling process is removed from the time-slice queue and its state is changed to waiting for process function; at the same time its process description is linked to the event queue of the activated auxiliary process. The latter serves calling processes one by one and returns them to the time-slice queue after the completion of each function.

Process functions are interruptable like other internal processes; however, from the viewpoint of calling processes they are indivisible, since they are only executed one by one in the order of their request by the auxiliary process, and the calling processes are delayed until the functions are completed.

7. RESOURCE CONTROL 7.

This chapter describes a set of monitor rules which enable a parent process to control the allocation of resources to its children.

7.1 Introduction 7.1

In the multiprogramming system the internal processes compete for the following limited resources:

computing time
primary storage
message buffers
process descriptions
peripheral devices
backing storage

Initially all resources are owned by the basic operating system, s. As a fundamental principle enforced by the monitor, a process can only allocate a subset of its own resources to a child process. These are returned to the parent process when the child is removed.

7.2 Time-Slice Scheduling 7.2

All running processes are allocated time slices in a cyclical manner. The interrupt frequency of the hardware interval timer, and hence the length of the time slice, is fixed at 25.6 milliseconds.

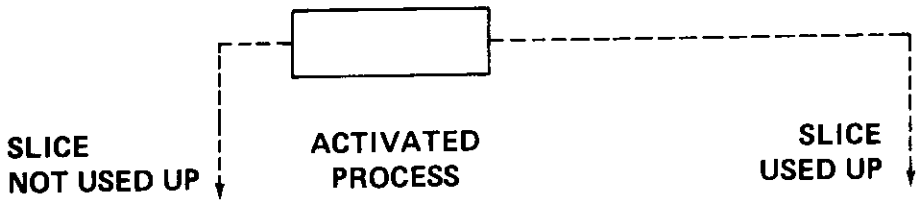
In practice internal processes often initiate input/output and wait for it in the middle of a time slice. This creates a scheduling problem when internal processes are activated by answers from external processes: Should the monitor link the activated process to the beginning or the end of the time-slice queue?

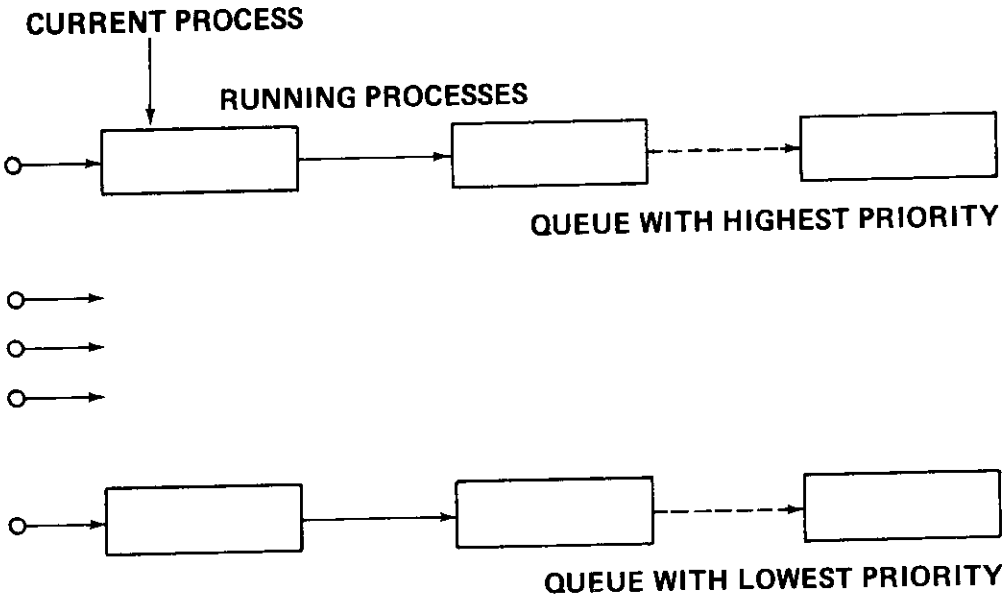
The first alternative ensures that processes can use devices at maximum speed, but it also presents the danger that a process can monopolize computing time by communicating frequently with fast devices. The second alternative prevents this, but introduces a delay in the time-slice queue, which slows devices down.

A modified form of round-robin scheduling has been introduced to solve this dilemma. As soon as a process is removed from the time-slice queue, the monitor stores the current value of the time quantum used by it. When the process is activated again, the monitor compares this value with the maximum time slice. So long as the time slice is not used up, the process is linked to the beginning of the queue; otherwise it is linked to the end of the queue and its time quantum is reset to zero.

This test is applied, and the time-slice queue is reorganized, whenever the monitor receives an interrupt, e.g. an external interrupt or a timer interrupt.

The monitor's round-robin scheduling attempts to share computing time evenly among active internal processes regardless of their position in the hierarchy. It permits a process to be activated immediately until it threatens to monopolize the central processor; only then is it thrust into the background to give other processes a chance. This is admittedly a built-in strategy on the microlevel. Parent processes can, in fact, only control the allocation of computing time to their children in relatively large portions (on the order of seconds), by means of the procedures start and stop internal process.





Over and beyond the round-robin scheduling of the monitor, one can assign priorities to internal processes. Each priority level has its own time-slice queue, in which processes run on a round-robin basis. When the queue with the highest priority is empty, i.e. when all of the processes are either waiting or stopped, process execution continues in the queue on the next highest level.

When internal processes are created, they all receive the same priority; the parent process, however, can change this priority by means of the procedure

```
set priority(name,priority level,result)
```

For accounting purposes the monitor retains the following information for each internal process: the time at which the process was created and the sum of the time quanta used by it; these quantities are denoted start time and run time.

7.3 Storage Allocation and Protection 7.3

An internal process can only create child processes within its own storage area. The monitor does not check whether the storage areas of child processes overlap. This freedom may be utilized, for example, to implement time-sharing of a common storage area among several processes.

The process description for an internal process contains three fields which define the rights of the process to read and write in the primary store. Two of the fields define the upper and lower limits of the storage area assigned to the process when it was created. Here the process may read and write. The third field defines the upper limit of a storage area where the process only may read. This area includes, among other things, common tables in the monitor.

An internal process may not read or write directly in the storage area of an internal process outside its own limits, but can send messages to request the exchange of data (by means of the monitor procedure copy).

7.4 Message Buffers and Process Descriptions 7.4

The monitor has room only for a finite number of message buffers, internal process descriptions, and area process descriptions.

These buffers and tables assume an identity only when they are actually used. Thus an internal process description, for example, is selected when an internal process creates another internal process, and released when the process is removed. So long as they remain unused, message buffers and process descriptions may be regarded as anonymous pools of resources.

It is therefore sufficient to specify the maximum number of each resource which an internal process may use. These so-called buffer, internal, and area claims are defined by the parent when a child process is created. The claims must be a subset of the parent's own claims, which are diminished accordingly; they are returned to the parent when the child is removed.

The buffer claim defines the maximum number of messages which an internal process may exchange simultaneously with other internal and external processes.

The internal claim limits the number of children which an internal process may have at the same time.

The area claim defines how many documents an internal process may access simultaneously.

The monitor decreases a claim by one each time a process actually uses one of its resources, and increases it by one when the resource is released. Thus at any moment the claims define the number of resources which may still be used by the process.

When an internal process sends a message, its buffer claim is decreased by one. When the receiver accepts the message, by calling wait message, its buffer claim is likewise decreased by one. When the receiver sends the answer, by means of send answer, its buffer claim is increased by one, and when the original sender accepts the answer, by means of wait answer, its buffer claim is also increased by one.

Thus two buffer claims, but only one physical message buffer are involved in a transfer of information. This enables the parent process, when it removes a child process, to retrieve immediately all of the claims which it originally allocated to the child. In other words, those message buffers in which the child had not had time to receive an answer before its removal are "borrowed" by the parent from the pool, which then incorporates them as answers are received.

7.5 The Right to Use a Given Device 7.5

As an external process is created when a name is assigned to a device (see Sect. 5.3), it also holds of peripheral devices that they assume an identity only when they are actually used for input/output. It would therefore seem natural to control the allocation of devices to internal processes by a complete set of claims - one for each kind of device.

In a system with remote devices, however, it is unrealistic to treat all devices of a given kind as a single, anonymous pool. An operating system must be able to force its children and their human operators to remain within a certain geographical configuration of devices.

It should be noted that the concept configuration must be defined in terms of physical devices rather than external processes, since a parent generally speaking does not know in advance which documents its children are going to use.

Configuration control is exercised in the following way: From the viewpoint of other processes, an internal process is identified by a name. Within the monitor, however, an internal process can also be identified by a single bit in a field. The external process descriptions include a field in which each bit indicates whether the corresponding internal process is a potential user of the device. Another field indicates the current user who has reserved the device in order to obtain exclusive access to the document.

Initially the basic operating system, s, is a potential user of all devices. A parent process can include or exclude a child as a user of any device, provided that the parent is also a user of the device, by means of the procedures

include user(child,device number,result)
exclude user(child,device number,result)

During the removal of a child, the monitor excludes it as a user of all devices.

All in all, three conditions must be fulfilled before an internal process can initiate input/output:

1. The device must be described as an external process with a unique name.
2. The internal process must be a potential user of the device.
3. The internal process must reserve the external process if it controls a sequential document.

7.6 Privileged Functions

Files on the backing store are described in a catalog, which is also kept on the backing store. In order to prevent an internal process from reserving an excessive amount of space in the catalog or on the backing store as such, the concept privileged monitor procedure has been introduced.

A parent process must provide each of its children with a function mask, in which each bit specifies whether the child is allowed to perform a certain monitor function. The mask must be a subset of the parent's own function mask.

At present these privileged functions include all monitor procedures which involve:

Handling of main and auxiliary catalogs.

Handling of auxiliary catalog entries.

Creation and removal of peripheral device names.

Initialization of the real-time clock.

8. STORAGE OF FILES AND OTHER MONITOR FEATURES

8.

This chapter is a survey of specific monitor features, viz. internal interruption, the real-time clock, and, especially, the storage of files on the backing store. (Another monitor feature, conversational access from consoles and terminals, is described in the RCSL publication Operating System s Reference Manual). Although these features are not essential primitive concepts, they are indispensable in practical multiprogramming systems.

8.1 Internal Interruption

8.1

The monitor can assist internal processes in the detection of infrequent events, e.g. violation of storage protection or arithmetic overflow. Such events cause an interruption of the internal process followed by a jump to an interrupt procedure within the process.

The interrupt procedure is defined by means of the monitor procedure

```
set interrupt(interrupt address,interrupt mask)
```

When an internal interrupt occurs, the state of the process (i.e. register values) is stored in the head of the interrupt procedure, and the monitor continues execution of the internal process in the body of the procedure:

```
interrupt address:  working registers
                    status register
                    instruction counter
                    interrupt cause
                    address register
                    (execution continues here)
```

The system distinguishes between the following causes of internal interruption:

protection violation
integer overflow
floating-point overflow or underflow
parameter error in monitor call
break forced by parent
parity error in store or on bus

The interrupt mask specifies whether arithmetic overflow should cause internal interruption. Other kinds of internal interrupts cannot be masked off.

If an internal process provokes an interrupt without having defined an interrupt procedure after its creation, the monitor removes the process from the time-slice queue and changes its state to running after error. This is also done if a parity error occurs in the store or on the bus. The process receives no more computing time in this state, but from the viewpoint of other processes it still exists. The parent of an erroneous process can, however, reactivate it by means of stop and start.

A parent can force a break in a child process as follows: First, stop the child; second, fetch the registers and interrupt address from the process description of the child and store the registers in the interrupt area together with the cause; third, modify the registers of the child to ensure that program execution will continue in the interrupt procedure; fourth, start the child again.

8.2 The Real-Time Clock

Real time is measured by means of a hardware interval timer, which counts modulo 32786 in units of 0.1 millisecond and interrupts the computer regularly every 25.6 milliseconds. Normally, however, the computer is interrupted much more frequently, viz. by peripheral devices each time a data transfer is completed. Whenever the monitor receives an interrupt, the time-slice queue is reorganized in accordance with the strategy described in Section 7.2.

The monitor uses the interval timer to update a programmed real-time clock of 48 bits. This clock can be initialized and sensed by means of the procedures

```
set clock(clock)
get clock(clock)
```

The setting of the clock is a privileged function. A standard convention adopted by operating systems (but not enforced by the monitor) is to have the clock express the time elapsed since midnight, December 31, 1967, in units of 0.1 millisecond.

The clock is an external process, which can accept messages from all internal processes. Such messages may specify, for example, that the process wishes to be removed from the time-slice queue for a certain interval. (See further the RCSL publication Interval Clock Process).

8.3 Storage of Files on the Backing Store

The monitor permits the storage of files on a backing store consisting of one or more physical volumes (e.g. disc packs). These in turn comprise one or more logical volumes, each described as a logical disc. The monitor makes the logical volumes appear as a single backing store with a number of 256-word segments.

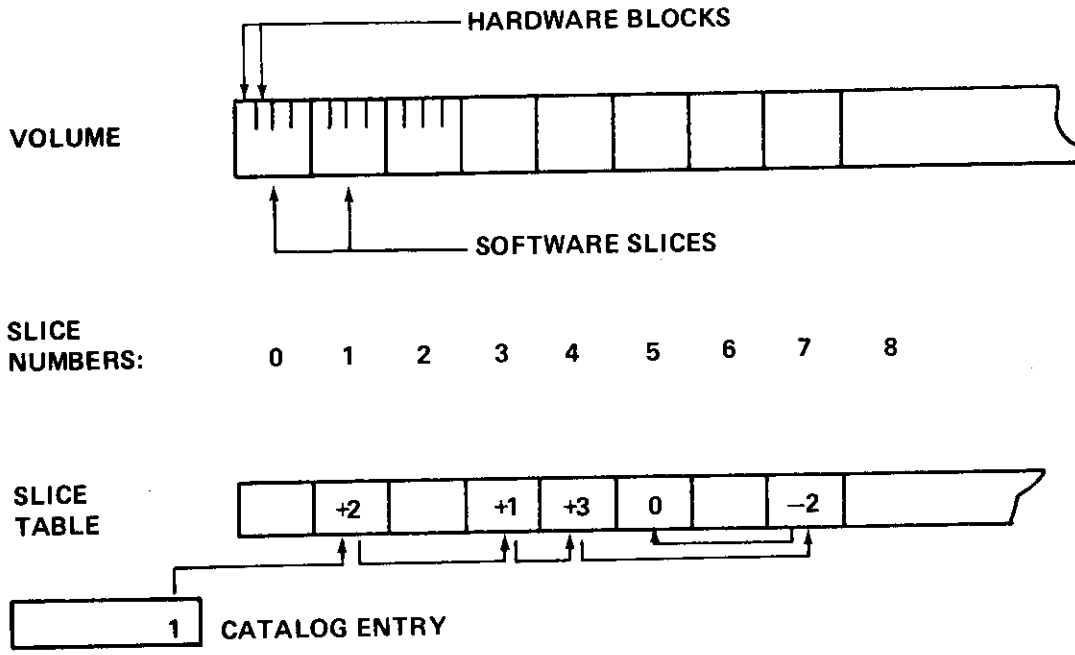
This logical backing store is organized as a collection of named files. Each file occupies a number of segments on a single logical volume. A fixed part of the logical volume is reserved for a volume catalog and a slice table, which together describe the files on the volume and their physical location.

The identification of a file requires a catalog search. In order to reduce the number of searches, input/output must be preceded by the explicit creation of an area process describing the file within the monitor. Area processes are treated as external processes by the internal processes; input/output is initiated by sending messages to the area processes specifying input/output operations, storage addresses, and relative segment numbers within the files.

8.3.1 File Structure, Slices, and the Slice Table

Each backing-storage volume is divided into hardware blocks of 256 24-bit words. The blocks are grouped in software slices of a fixed size, selected for each disc according to its physical characteristics. The slice size may therefore vary from volume to volume. The set of slices for a given volume is mapped onto a slice table containing a 12-bit halfword for each slice.

A file on the volume consists of a whole number of slices, with the corresponding halfwords in the slice table chained together and terminated by zero. In the figure below, halfwords 1, 3, 4, 7, and 5 of the slice table are chained together and represent a file of five slices. The numbers in the halfwords, e.g. +2, point to the next halfword in the chain.



The number of the first slice is defined in the catalog entry describing the file. Free slices have a unique value in the slice table. When a file is extended, a free slice-table halfword is linked to the end of the chain representing the file. Once a volume is mounted, its slice table is kept in the primary store for fast reference. Whenever a permanent file is extended or reduced, the slice table is copied to the volume as a safety measure.

8.3.2 Catalogs and Catalog Entries

The catalog is a fixed area on the volume divided into a number of entries identified by names.

The name assigned to a catalog entry has two parts: the name proper (up to 11 ISO characters) and a name base, which is represented by two integers. The name base determines which internal processes may read and write in the file described by the entry (see Sect. 8.3.4).

Each file is described by an entry in the main catalog, which is simply a file on the principal backing-storage volume. Each entry contains the following information in 17 words:

number of the first slice
permanency of the file
file name and its hash key
hierarchical position of the file
file size in segments
name of the logical volume containing the file
optional parameters for the internal process

Note that the volume name may refer to, say, a reel of magnetic tape instead of a disc pack; in this case the file size is negative and defines the kind of device in question, and no backing-storage area is reserved.

The volume catalog on the logical volume, which is called the auxiliary catalog, is a file with a structure like that of the main catalog.

Each catalog entry contains a 3-bit permanence key, which assists the operating systems in their administration of free backing-storage space.

The monitor checks only whether the value of the permanence key is greater than that of the minimum auxiliary catalog key (a monitor option), in which case the file is described in the auxiliary as well as the main catalog. It rests with the operating system to delete temporary files when it removes the child process that created them. (Thus the BOSS operating system, for example, does this, but the basic operating system, s, does not). Temporary files are automatically deleted when the system is shut down, whereas permanent files are deleted only at the request of the user according to the rule of file change (see Sect. 8.3.3).

Thus the monitor, in accordance with the system design philosophy, provides only the necessary mechanism for the handling of temporary files, but leaves the actual strategy of file deletion to the hierarchy of processes, i.e. to the operating systems.

An internal process can ensure the survival of a catalog entry and the file which it describes by means of the monitor procedure

```
permanent entry(name,permanence key,result)
```

An entry is created by calling the procedure

```
create entry(name,tail,result)
```

The identifying name is contained in the head of the entry, whereas the remaining information is found in the tail.

When an entry and a file are created, the monitor sets the name base of the entry equal to the catalog base of the creating process (see Sect. 6.1).

Internal processes can look up, change, rename, or remove existing entries by means of the procedures

```
look up entry(name,tail,result)
change entry(name,tail,result)
rename entry(name,new name,result)
remove entry(name,result)
```

When the system is started, all entries from all volumes are collected in a single catalog, viz. the main catalog, which describes itself in an entry named <:catalog:>.

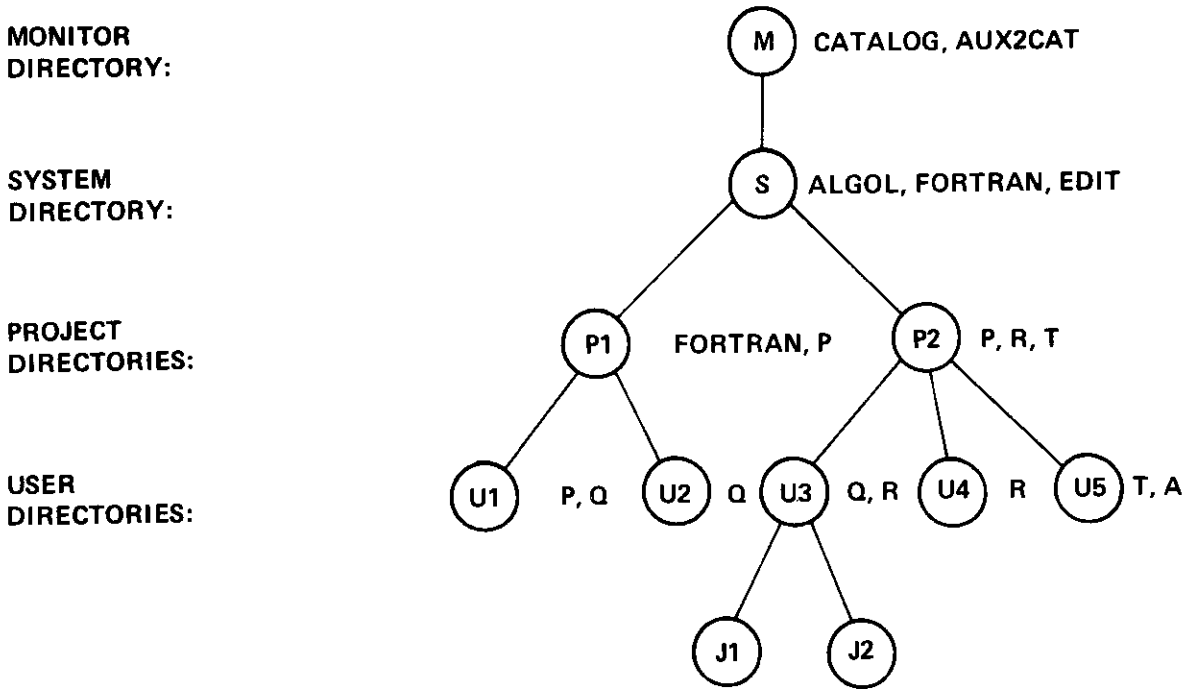
The search for catalog entries is minimized by using a hashed value of names to define the first segment to be examined. Each segment contains 15 entries; thus most catalog searches require only the input of a single segment unless the catalog is brimful.

8.3.3 The "Directory Hierarchy" and Read/Write Protection 8.3.3

Catalog entries and the files which they describe may be visualized as a hierarchy of directories, albeit the latter do not exist as separate structures within the system; a "directory" is simply a set of files with a given name base (see Sect. 8.3.4).

An internal process (job) selects a directory by setting its own catalog base equal to the name base of the relevant files.

In the following figure, the files in each directory are indicated on the right, e.g. algol. The directory names, e.g. s, are for explanatory purposes only.



Each internal process selects a "local directory" in which files are looked for first. If the file name is not found here, the directory on the next higher level is searched, and so on. This is the search rule for file names.

Take as an example user 3 in the figure above. His process has u3 as its local directory, and if he looks up fortran, u3 will be searched first, then p2, and finally s, which contains the desired entry. If user 2 looks up fortran, the desired entry is found in p1, and thus user 2 will access a different FORTRAN compiler. (Perhaps users 1 and 2 are developing a new version of FORTRAN).

Within certain limits a process may select another local directory in which files are looked for first. User 3 will typically be allowed to select u3 or p2 as a local directory. When he selects the latter, u3 files q and r are invisible and p2 files p, r, and t appear.

User 3 may also select a local directory corresponding to a branch from u3, viz. j1 or j2. When a new file is created, it appears in the local directory, and thus user 3 may build up new directories (j1 and j2).

The limitations on selecting a local directory are as follows: Each process has a "standard directory" (u3 in the case of user 3) and a "maximum directory" (p2 in the case of user 3). The process may select a local directory either on the path from the standard directory to the maximum directory or on a branch from the standard directory. This is the rule of scope transition.

Write Protection

Write protection is implemented according to the following rule of file change: A process may change a file or catalog entry if the file is in a directory which could be a local directory for the process. Thus user 3 is only allowed to change files in u3, p2, and a branch from u3.

Let us assume that users 1 and 2 are the maintenance group and can change files in the system directory. Thus user 1 should have s as his maximum directory and u1 as his standard directory. This allows him to select u1, p1, and s (and a branch from u1) as his local directory. The rule of scope transition ensures that he cannot change files in m, p2, u2, u3, u4, or u5.

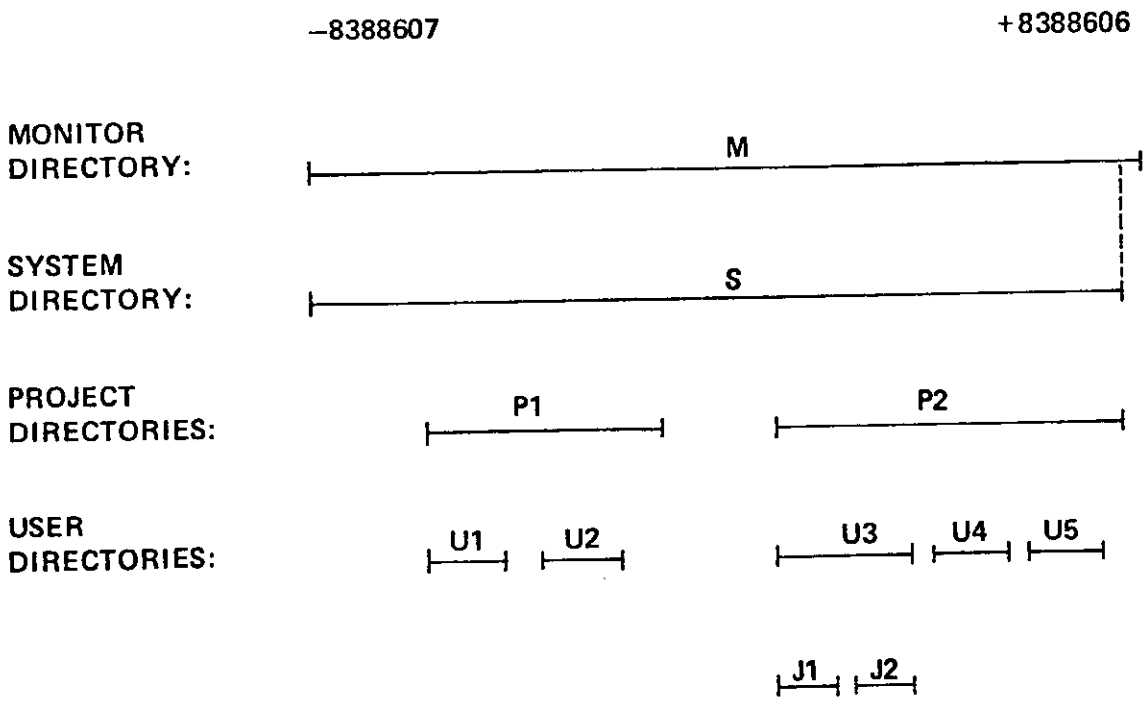
Read Protection

Read protection is implemented by the search rule for file names in conjunction with the limitations on selecting a local directory. Thus user 3, for example, cannot read or look up a file in u4, u5, u1, u2, or p1. Even user 1, who can change system files, cannot read files in u2, u3, u4, u5, or p2.

An operating system will typically have s as both its standard directory and its maximum directory. Thus it can select p1, u1, and so on as local directories and change or read files there; but it still cannot change files in m (the main and auxiliary catalogs).

8.3.4 Implementation of the "Directory Hierarchy"

The hierarchical position of a file is defined by its name base (see Sect. 8.3.2). The name base of a file on a low level is inside the name bases of the files above it. In the following figure, the name bases of the files in the previous figure are shown. The scale is not linear. The length of p1 and p2 is typically 100, that of u1, u2, u3, u4, and u5 10.



It has already been stated that a "directory" is simply a set of files with a given name base. Within the monitor these sets of files are implemented by means of the various bases that are associated with files and internal processes. Thus a "project directory" from the viewpoint of a process is expressed by its max base and a "user directory" by its current catalog base.

When an internal process (operating system) creates a child process, it defines the max base and the standard base of the child as being inside or equal to its own max base and standard base, i.e. the child may not see files which its parents cannot see.

The monitor then sets the name base of the child equal to the catalog base of its parent and the catalog base of the child equal to the standard base defined for it by its parent.

The child process may now "select a local directory," i.e. define its own catalog base as currently equal to the name base of the files in question, according to the rule of scope transition.

Implementation of the Rule of Scope Transition

The catalog base of an internal process must be inside or equal to its standard base or surround or be equal to its standard base and inside or equal to its max base.

Implementation of the Search Rule for File Names

When an internal process looks up a file named f, all files named f are searched for in the main catalog. Of the files which have a name base surrounding or equal to the currently defined catalog base of the process, the file with the smallest name base is selected.

Implementation of the Rule of File Change

1. If the name base of a file surrounds or equals the catalog base of an internal process and is inside or equal to its max base, the process may create an area process for read/write access to the file.
2. If the name base of a file surrounds the max base of an internal process, the process may create an area process for read access to the file.
3. If the name base of a file does not surround or equal the catalog base of an internal process, the process may not create an area process for access to the file.

When an internal process has created an area process and used it once, the internal process may redefine its catalog base.

8.3.5 Area Processes

In order to be used for input/output, a file must be looked up in the catalog and described as an area (i.e. external) process within the monitor:

```
create area process(name,result)
```

The area process is created with the same name as the catalog entry.

Following this, internal processes can send messages with the following format to the area process:

```
message:  input/output operation
          first storage address
          last storage address
          first relative segment
```

The reader is reminded that the tables used to describe area processes within the monitor are a limited resource, which is controlled by means of area claims defined by the parent process (see Sect. 7.4).

The backing store is a random-access medium, which serves as a common data bank. In order to utilize this property fully, internal processes should be able to input simultaneously from the same file (e.g. when copies of a compiler are executed in parallel). On the other hand, access to a file should be exclusive during output, because its contents are undefined from the viewpoint of other processes.

A distinction is therefore made between internal processes that are potential users of an area process and the single process that may have reserved the area process exclusively. This distinction was also made for peripheral devices (see Sect. 5.2), but now the rules of access are different: An internal process is a user of an area process after the creation of it, which enables the process to perform input so long as no other process makes a reservation. An internal process can reserve an area process, if it can change the file which the area process refers to. After reservation the process can perform both input and output.

Finally we should mention that the catalog is described permanently as an area process within the monitor. This enables internal processes to input and scan the catalog sequentially, e.g. during the detection and removal of temporary entries. Only the monitor itself, however, can perform output to the catalog.

RELATED PUBLICATIONS

Note that the RCSL numbers and edition dates of the following publications are subject to change.

An Introduction to RC8000 Operating Systems,
RCSL No. 31-D552, December 1979

Interval Clock Process,
RCSL No. 31-D530, December 1978

Operating System s Reference Manual,
RCSL No. 31-D455, June 1978

RC8000 Computer Family Reference Manual,
RCSL No. 42-i 1235, June 1979

RC8000 Monitor, Part 2, Reference Manual,
RCSL No. 31-D477, January 1978

RC8000 Monitor, Part 3, External Processes,
RCSL No. 31-D478, January 1979

RCNET, General Information,
RCSL No. 43-Ri0635, December 1976

GLOSSARY

Activate process (to)

To link an internal process to the time-slice queue in order to make it running.

Area process

Input/output of a file on the backing store identified by name.

Base

An interval represented by two integers. The bases assigned to an internal process determine which other internal processes can see it as well as which backing-storage files it may access.

Catalog

A fixed part of the backing store divided into named entries. An entry can describe a file on the backing store or some other document.

Child process

An internal process created by another internal process, which is described as its parent in the process hierarchy.

Create process (to)

To create a table within the monitor describing a process by its name, kind, resources, event queue, and current state.

Delay process (to)

To remove an internal process temporarily from the time-slice queue in order to make it wait for an event outside the process.

Document

A physical medium on which a specific collection of data is stored, e.g. a roll of paper tape, deck of punched cards, printer form, reel of magnetic tape, or file on the backing store.

Event queue

The queue in which a process receives messages and answers from other processes.

External process

A general term for an area process or a peripheral process.

Internal interrupt

An interruption of an internal process caused by protection violation, arithmetic overflow, an erroneous monitor call, or the parent of the process.

Internal process

The execution of one or more interruptable programs in a contiguous storage area identified by name.

Link

The logical data path between the peripheral process description within the monitor and the physical peripheral device. The link concept implies that an internal process can access any device without being aware of its geographical location.

Monitor

A resident program with complete control of storage protection, input/output, and interrupts. It contains descriptions of all processes and controls the sharing of computing time among them. It also contains procedures which internal processes can call in order to create and control other processes and communicate with them.

Multiprogramming

Simultaneous execution of several programs loaded in the store by multiplexing of the central processor controlled by timer interrupts.

Operating system

A program that controls the scheduling and resource allocation of other programs in order to obtain a specific mode of operation, e.g. batch processing, real-time scheduling, or time-sharing. During execution an operating system is synonymous with a parent process.

Parent process

An internal process that creates and controls another internal process, which is described as its child in the process hierarchy.

Peripheral process

Input/output or interrupt signals of a peripheral device identified by name. It usually involves the use of a specific document mounted on the device.

Process hierarchy

A family tree describing the control relationships among internal processes, which are called ancestors, parents, children, or descendants according to their position in the hierarchy relative to a given process. An internal process can only start, stop, or remove its own child processes and their descendants.

Program

A collection of instructions specifying a computational process.

Pseudo process

A description of an internal or external process, which permits that process to appear under a pseudonym. All messages sent to a pseudo process are forwarded to the internal process that created it.

Remove process (to)

To remove a table within the monitor describing a process by its name, kind, resources, event queue, and current state.

Resources

A general term for the amount of computing time, primary storage, message buffers, process descriptions, peripheral devices, and backing storage allocated to an internal process.

Running process

An internal process in the time-slice queue which is executing instructions or ready to do so.

Start process (to)

To activate an internal process at the request of its parent.

Stop process (to)

To delay an internal process at the request of its parent.

Time-slice queue

A queue of internal processes that share computing time in a cyclical manner.

Waiting process

An internal process that has been removed temporarily from the time-slice queue in order to wait for an event outside the process.

INDEX

RETURN LETTER

Title: RC8000 Monitor, Part 1 RCSL No.: 31-D476

A/S Regnecentralen af 1979/RC Computer A/S maintains a continual effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback, your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

Do you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Name: _____ Title: _____

Company: _____

Address: _____

Date: _____

Thank you

42-i 1288

. . . . . Fold here . . . . .

. . . . . Do not tear - Fold here and staple . . . . .

Affix
postage
here

REGNECENTRALEN
af 1979

Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark