RC 4000 SOFTWARE: MULTIPROGRAMMING SYSTEM

PER BRINCH HANSEN

(1969)

The RC 4000 multiprogramming system 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. This manual defines the functions of the monitor and the basic operating system, which allows users to initiate and control parallel program execution from typewriter consoles. The excerpt reprinted here is the general description of the philosophy and structure of the system. This part will be of interest to anyone wishing an understanding of the system in order to evaluate its possibilities and limitations without going into details about exact conventions. The discussion treats the hardware structure of the RC 4000 only in passing.

1 SYSTEM OBJECTIVES

This chapter outlines the philosophy that guided the design of the RC 4000 multiprogramming system. It emphasizes the need for different operating systems to suit different applications.

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

P. Brinch Hansen, *RC 4000 Software: Multiprogramming System*, Part I General Description. Regnecentralen, Copenhagen, Denmark, April 1969, 13–52.

The present system is implemented on the RC 4000 computer, a 24-bit, binary computer with typical instruction execution times of 4 microseconds. It permits practically unlimited expansion of the internal store and standardized connection of all kinds of peripherals. 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 internal store of 16 k words backed by a fast drum or disk. Programs can be written in any of the available programming languages and contain programming errors. The storage protection system guarantees noninterference among 8 parallel programs, but it is possible to start up to 23 programs provided some of them are error free.

The system uses standard multiprogramming techniques: the central processor is shared between loaded programs. Automatic swapping of programs in and out of the store is possible but not enforced by the system. Backing storage is organized as a *common data bank*, in which users can retain named files in a semi-permanent manner. The system allows a *conversational mode* of access from typewriter consoles.

An essential part of any multiprogramming system is an *operating system*, 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 the 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, a vital requirement of any operating system is that it allows him to change the mode of operation it controls; otherwise his freedom of design can be seriously limited. Unfortunately this is precisely what present operating systems do not allow. Most of them are based exclusively on a single mode of operation, such as batch processing, priority scheduling, real-time scheduling, or time-sharing.

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

This unfortunate situation indicates that the main problem in the design of a multiprogramming system is not to define functions that satisfy specific

operating needs, but rather to supply a system nucleus that can be extended with new operating systems in an orderly manner. This is the primary objective of the RC 4000 system.

The nucleus of the RC 4000 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 RC 4000 more 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 among running programs,
>
> initiation of data transfers to or from peripherals.

The monitor has no built-in strategy of program execution and resource allocation; it allows any program to initiate other programs in a hierarchal 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 in the system as other programs without modification of the monitor. Furthermore operating systems can be replaced dynamically, enabling each installation to switch among various modes of operation; several operating systems can, in fact, be active simultaneously.

In the following chapters we shall explain this dynamic operating system concept in detail. In accordance with our philosophy all questions about particular strategies of program scheduling will be postponed, and the discussion will concentrate on the fundamental aspects of the control of an environment of parallel processes.

2 ELEMENTARY MULTIPROGRAMMING PROBLEMS

This chapter introduces the elementary multiprogramming problems of mutual exclusion and synchronization of parallel processes. The discussion is restricted to the logical problems that arise when independent processes try to access common variables and shared resources. An understanding of these concepts is indispensable to the uninitiated reader, who wants to appreciate the difficulties of switching from uniprogramming to multiprogramming.

2.1 Multiprogramming

In multiprogramming the sharing of computing time among programs is controlled by a *clock*, which interrupts program execution frequently and activates a *monitor program*. The monitor saves the registers 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 is only able to execute one instruction at a time, multiprogramming creates the illusion that programs are being executed simultaneously, mainly because peripherals assigned to different programs indeed operate in parallel.

2.2 Parallel Processes

Most of the elementary problems in multiprogramming arise from the fact that one *process* (e.g. an executed program) cannot make any assumptions about the relative speed and progress of other processes. This is a potential source of conflict whenever two processes try to access a common variable or a shared resource.

It is evident that this problem will exist in a truly parallel system, in which programs are executed simultaneously on several central processors. It should be realized, however, that the problem will also appear 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.

Another way of stating this is that if one considers the system as seen from within a program, it is irrelevant whether multiprogramming is implemented on one or more central processors—the logical problems are the same.

Consequently a multiprogramming system must in general be viewed as an environment with 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 illustrated abundantly in the following chapters.

2.3 Mutual Exclusion

The idea of multiprogramming is to share the computing equipment among a number of parallel programs. At any moment, however, a given resource

must belong to one program only. In order to ensure this it is necessary to introduce global variables, which programs can inspect to decide whether a given resource is available or not.

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

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

While this is taking place the program may be interrupted after the loading of the boolean, but before inspection and assignment to it. The register containing the value of the boolean is then stored within the monitor, and program q is started. Q may load the same boolean and find that the typewriter is available. Q accordingly assigns the value false to the boolean and starts using the typewriter. After a while q is interrupted, and at some later time p is restarted with the original contents of the register reestablished by the monitor. Program p continues the inspection of the original value of the boolean and concludes erroneously that the typewriter is available.

This conflict arises because programs have no control over the interrupt system. Thus the only indivisible operations available to programs are single instructions such as load, compare, and store. This example shows that one cannot implement a multiprogramming system without ensuring a *mutual exclusion* of programs during the inspection of global variables. Evidently 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 disabled mode.

In the use of 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:

```
process p: wait and reserve(r) ... wait and reserve(s) ...
process q: wait and reserve(s) ... wait and reserve(r) ...
```

This can cause both processes to wait forever, since neither is aware of that it wants what the other one has.

To avoid this problem we need a third process (an *operating system*) that 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 other until resources become available).

2.4 Mutual Synchronization

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 also 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 at the very moment the sender wishes to deliver it, or conversely the receiver can become idle at a time when there is no further 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* on its own request and *activated* on request from another process.

For a more extensive analysis of multiprogramming fundamentals, the reader should consult E. W. Dijkstra's monograph: *Cooperating Sequential Processes*. Math. Dep. Technological University, Eindhoven, (Sep. 1965).

3 BASIC MONITOR CONCEPTS

This chapter opens a detailled description of the RC 4000 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

The aim has been to implement a multiprogramming system that 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 safety problems that arise when erroneous or malicious processes try to interfere with other processes. They do, however, leave the choice of particular strategies of program scheduling to the processes themselves.

With this objective in mind we have implemented the following fundamental mechanisms within the monitor:

simulation of parallel processes,
communication among processes,
creation, control, and removal of processes.

3.2 Programs and Internal Processes

As a first step we shall 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 RC 4000.

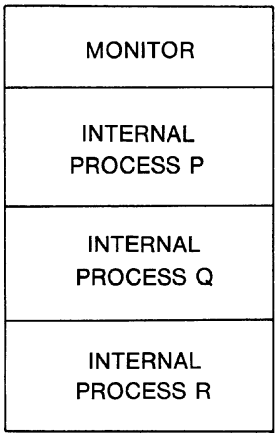
We distinguish between internal and external processes, roughly corresponding to program execution and input/output.

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

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

Later it will be explained how internal processes are created and how programs are loaded into them. At this point it should only be noted that an internal process occupies a fixed, contiguous storage area during its whole lifetime. The monitor has a *process description* of each internal process; this table defines the name, storage area, and current state of the process.

Computing time is shared cyclically among all active internal processes; as a standard the monitor allocates a maximum time slice of 25 milliseconds to each internal process in turn; after the elapse of this interval the process is interrupted and its registers are stored in the process description; following this the monitor allocates 25 milliseconds to the next internal process, and so on. The cyclic queue of active internal processes is called the *time slice queue*.



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

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

3.3 Documents and External Processes

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

A *peripheral device* is an item of hardware connected to the data channel and identified by a device number.

A *document* is a collection of data stored on a physical medium. Examples of documents are:

a roll of paper tape,
a deck of punched cards,
a printer form,
a reel of magnetic tape,
a data area on the backing store.

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 it uses.

For each external process the monitor keeps a process description defining its name, kind, device number, and current state. The *process kind* is an integer defining the kind of peripheral device on which the document is mounted.

For each kind of external process the monitor contains an interrupt procedure that can initiate and terminate input/output on request from internal processes.

3.4 Monitor

The monitor is a program activated by means of interrupts. It can execute privileged instructions in the disabled mode, meaning that (1) it is in complete control of input/output, storage protection, and the interrupt system, and that (2) it can execute a sequence of instructions as an indivisible entity.

After initial system loading the monitor resides permanently in the internal store. We do not regard the monitor as an independent process, but rather as a software extension of the hardware structure, which makes the computer more attractive for multiprogramming. Its function is to (1) keep descriptions of all processes; (2) share computing time among internal and external processes; and (3) implement procedures that processes can call in order to create and control other processes and communicate with them.

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

4 PROCESS COMMUNICATION

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 safety and efficiency.

4.1 Message Buffers and Queues

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 a 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 among internal processes:

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

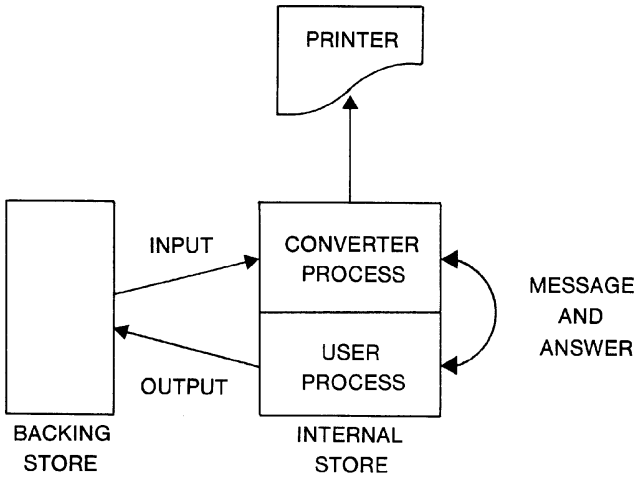
Send message copies a message into the first available buffer within the pool and delivers it in the queue of a named receiver. The receiver is activated if it is waiting for a message. The sender continues after being informed of the address of the message buffer.

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-existing process.

The use of these procedures can 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 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 line printer is shared by all user processes with a minimum delay.



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

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

```
user process:
   ...
   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 bottleneck in the system, however, a conversational process must be prepared to be actively engaged in more than one conversation at a time. As an example think of a conversational process that engages itself, on request from another process, in a conversation with one of several human operators in order to perform some manual operation (mounting of a tape etc.). If one restricts a conversational process to only accepting one request (i.e. a message) at a time, and to completing the requested action before receiving the next request, the unacceptable consequence of this is that other processes (including human operators at consoles) can have their requests for response delayed for a long or even undefined time.

As soon as a conversational process has started 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 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) can be tied up by already initiated actions; thus in some cases the process will not be able to honor new requests before old ones are completed. In this case the process wants 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.

Consequently we have introduced two more general communication procedures, 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 an answer. In accordance with this the queue of a process from now on will be called the *event queue*.

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 last buffer address is zero, the queue is examined from the start. The procedure does not remove the next buffer from the queue or in any other way change its status.

As an example, consider an event queue with two pending buffers A and B:

queue = buffer A, buffer B

The monitor 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; while 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 the buffer 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.

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
exam request:    if resources not available then go to next event;
init action:     get event(buffer);
                 reserve resources;
                 ...
                 send message to some other process;
                 save state of action;
                 end else
                 begin comment: result = answer;
term action:     restore state of action;
                 get event(buffer);
                 release resources,
                 send answer to original sender;
                 end;
                 go to first event;
```

The process starts by examining its queue; if 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 resources, and initiates an action. As soon as this involves the sending of a message to some other process, the conversational process saves information about the state of the incomplete action and proceeds to examine its queue from the start 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 on request from typewriter consoles. S can be engaged in conversations with several consoles at the same time. It will only postpone an operator request if its storage is occupied by other requests, or if it is already in the middle of an action requested from the same console.

4.4 Advantages of Message Buffering

In the design of the communication scheme we have given full recognition to the fact that the multiprogramming system is a dynamic environment, in which some of the 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 about the existence of other processes. This is reflected in the procedure *wait message*, which makes it possible for a process to be unaware of the existence of other processes until it receives messages from them.

On the other hand once a communication has been established between two processes (e.g. by means of a message), they need a common identification of it 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 an identification of a conversation.

A happy consequence of this is that it enables two processes to exchange more than one message at a time. We must be prepared for the occurence 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 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 delivery of a message or an answer regardless of whether the receiver is ready to process it or not.

In order to make the system dynamic it is vital that a process can be removed at any time, even if it is engaged in one or more conversations. In 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 sensible to remove a user process that has completed its task and is now only waiting for an answer from the converter process. In this case the monitor leaves all messages from the 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 situation is also possible: during the removal of a process, the monitor finds 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 process that does not exist.

The main drawback of message buffering is that it introduces yet another

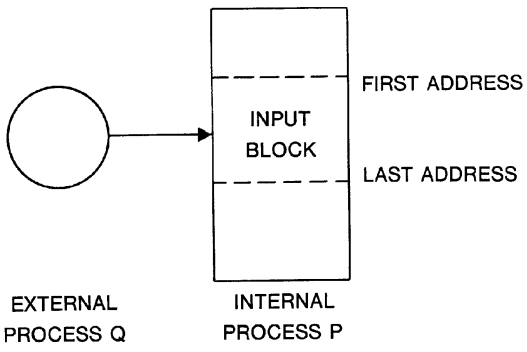
resource problem, since the common pool contains a finite number of buffers. If a process was allowed to empty the pool by sending messages to ignorant processes, which do not respond with answers, further communication within the system would be blocked. We have consequently set a limit to the number of messages a process can send simultaneously. By doing this, and by allowing a process to transmit an answer in a received buffer, we have placed the entire risk of a conversation on the process that opens it (see Section 7.4).

5 EXTERNAL PROCESSES

This chapter clarifies the meaning of the external process concept. It explains initiation of input/output by means of messages from internal processes, dynamic creation and removal of external processes, and exclusive access to documents by means of reservation. The similarity of internal and external processes is stressed.

5.1 Initiation of Input/Output

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



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. Following this it uses the kind parameter in the process description of process q to switch to a piece of code common to all magnetic tapes. If the tape station is busy, the message is merely left in its 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 delivers it in the queue of p, which in turn receives it by calling *wait answer*:

wait answer(result, answer, buffer)

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

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

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

Essentially all external processes follow this scheme, which can be defined by the following algorithm:

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

With low-speed, character-oriented devices, the monitor repeats input/output and the interrupt response for each character until a complete block has been transferred; (while this is taking place, the time between interrupts is of course shared among internal processes). Internal processes can therefore regard all input/output as block oriented.

5.2 Reservation and Release

The use of message buffering provides a direct way 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 their order of arrival. An example of this is the use of a single typewriter for output of messages to a main operator. This method of sharing a device ensures that a block of data is input or output as an indivisible entity. When sequential media such as paper tape, punched cards, or magnetic tape are used, however, an internal process must have exclusive access to the entire document. This is obtained by calling the following monitor procedure:

reserve process(name, result)

The result indicates whether the reservation has been accepted or not. An external process that handles sequential documents of this kind rejects messages from all internal processes except 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, however, also do this explicitly 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 must, however, 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 correct 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 has, in fact, no way of ensuring whether a given document is mounted on a device. Furthermore, there are some devices which operate without documents, e.g. the real-time clock.

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

remove process(name, result)

It is also possible to implement an automatic removal of the process name when the monitor detects operator intervention in a device. At present, this is done only in connection with magnetic tapes (see Section 10.1).

5.4 Replacement of External Processes

The decision to control input/output by means of interrupt procedures within the monitor, instead of using dedicated internal processes for each kind of peripheral device, was made to obtain immediate initiation of input/output after the sending of messages. In contrast the activation of an internal process merely implies that it is linked to the time slice queue; after activation several time slices can elapse before the internal process actually starts to execute instructions.

The price paid for the present implementation of external processes is a prolongation of the time spent in the disabled mode within the monitor. This limits the system's ability to cope with real-time events, i.e. data that are lost unless they are input and processed within a certain time.

An important consequence of the uniform handling of internal and external processes is that it allows us to replace any external process by an internal process of the same name; other processes that communicate with it are quite unaware of this replacement.

Thus it is possible to improve the response time of the system by replacing a time-consuming external process, such as the paper tape reader, by a somewhat slower internal process, which executes privileged instructions in the enabled mode.

This type of replacement also makes it possible to enforce more complex rules of access to a document. In the interests of security, for example, one might want to limit the access of an internal process to one of several files recorded on a particular magnetic tape. This can be ensured by an internal process that traps all messages to the tape and decides whether they should be passed on to it.

As a final example let us consider the problem of debugging a process control system before it is connected to an industrial plant. A convenient way of doing this is to replace analog inputs with an internal process that simulates relevant values of actual measuring instruments.

We conclude that the ability to replace any process in the system with another process is a very useful tool. This can now be seen as a practical

result of the general, but somewhat vague idea (expressed in Section 2.2) that internal and external processes are independent processes, which differ only in their processing capability.

6 INTERNAL PROCESSES

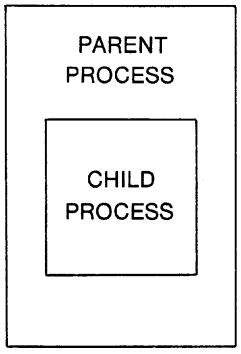
This chapter explains the creation and control of internal processes. The emphasis is on the hierarchal structuring of internal processes, which makes it possible to extend the system with new operating systems. The dynamic behaviour of the system is explained in terms of process states and the transition between these.

6.1 Creation, Control, and Removal

Internal processes are *created* on request from other internal processes by means of the monitor procedure:

create internal process(name, parameters, result)

The monitor initializes the process description of the new internal process with its name and storage area selected by the *parent process*. The storage area must be within the parent's own area. Also specified by the parent is a protection key, which must be set in all storage words of the *child process* before it is started.



After 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 typewriter console it can use for operator communication.

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

start internal process(name, result)

which sets the protection keys within the child and links it to the time slice queue. The child now shares time slices with other active processes including the parent.

On request from a parent process, the monitor waits for the completion of all input/output initiated by a child process and *stops* it, i.e. removes it from the time slice queue:

stop internal process(name, buffer, result)

The meaning of the message buffer will be made clear 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)

During removal, the monitor generates dummy answers to all messages sent to the child and releases all external processes used by it. Finally the protection keys are reset to the value used within the parent process. The parent can now use the storage area to create other child processes.

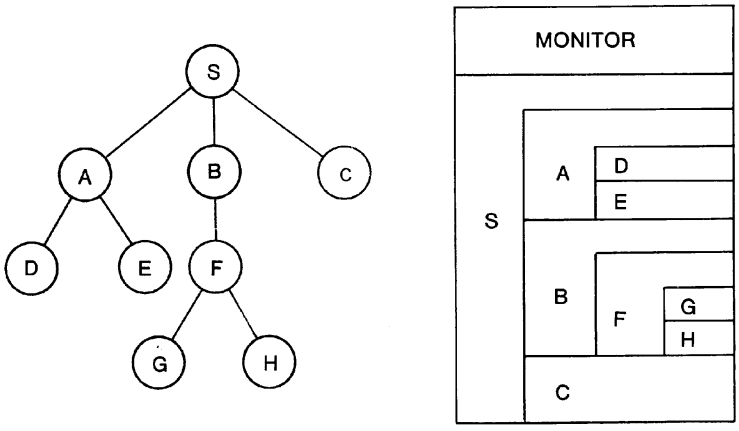
6.2 Process Hierarchy

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 fundamental set of procedures allows the

dynamic creation and control of processes as well as communication among them.

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

After initial loading the internal store contains the monitor and an internal process, s, which is the *basic operating system.* S can create parallel processes, a, b, c, etc., on request from consoles. These processes can in turn create other processes, d, e, f, etc. Thus while s acts as a primitive operating system for a, b, and c, these in turn act as operating systems for their children, d, e, f, etc. 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 of the total number or processes. At present the maximum number of internal processes is 23 including the basic operating system s. It must, however, be remembered that the storage protection system only provides mutual protection of 8 independent processes. When this number is exceeded, one must rely on some of the processes being error free.

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

As regards the future 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 and Fortran languages for the RC 4000 contain facilities for calling the monitor and initiating parallel processes. Thus it is possible to write operating systems in high-level languages.

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

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

6.3 Process States

We are now in a position to define the possible states of an internal process as described within the monitor. An understanding of the transition from one state to the other is vital as a key to the dynamic behaviour 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 either be *waiting* for a *message*, an *answer*, or an *event*, as explained in Chapter 4.

Of a more complex nature are the situations in which a process is *waiting*

to be *stopped* or *started* by another process. In order to explain this we shall once more refer to the family tree shown in the previous section.

Let us say that process b wants to stop its child f. The purpose of doing this is to ensure that all program execution and input/output within the storage area of process f is stopped. Since a part of the 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 already have been stopped by their own parents. In the present example process g may still be running, while process h may have been stopped by its parent f. Consequently the monitor should only stop processes f and g.

Consider now the reverse situation, in which process b starts its child f again. Now the purpose is to reestablish the situation exactly as it was before process f was stopped. Thus the monitor must be very careful only to start those descendants of f that were stopped along with f. In our example the monitor must start processes f and g but not h. Otherwise we confuse f, which still relies on its child h being stopped.

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

The possible *states* of an internal process are the following:

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 meaning of the state *running after error* is explained in Section 8.1.

When a parent wants 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 arrival of an answer from an external process, 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 involved processes are waiting for start is the stop operation finished. This can last some time, and it may not be acceptable to the parent (being an operating system with many other duties) to be inactive for so long. For this reason the stop operation is split 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 procedures *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 an event, its state will be changed to waiting for stop, as explained above, but at the same time its instruction counter is decreased by two in order that it can, repeat the call of *wait message*, *wait answer*, or *wait event* when it is started again.

It should be noted that a process can receive messages and answers in its queue in any state. This ensures that a process does not loose contact with its surroundings while stopped.

The meaning of the state *waiting for process function* is explained in Section 9.1.

7 RESOURCE CONTROL

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

7.1 Introduction

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

computing time
storage and protection keys
message buffers
process descriptions
peripheral devices
backing storage

Initially all resources are owned by the basic operating system s. As a basic 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

All running processes are allocated *time slices* in a cyclical manner. Depending on the interrupt frequency of the hardware interval timer, the length of a time slice can vary between 1.6 and 1638.4 milliseconds. A reasonable time slice is 25.6 milliseconds; with shorter intervals the percentage of computing time consumed by timer interrupts grows drastically; with longer intervals the delay between activation and execution of an internal process increases.

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: Should the monitor link processes to the beginning or to the end of the time slice queue? The first possibility ensures that processes can use peripherals with maximum speed, but there is the danger that a process can monopolize computing time by communicating frequently with fast devices. The second choice prevents this, but introduces a delay in the time slice queue, which slows down peripherals.

We have introduced a modified form of round-robin scheduling to solve this dilemma. As soon as a process is removed from the time slice queue, the monitor stores the actual value of the *time quantum* used by it. When the process is activated again, the monitor compares this quantum with the maximum time slice. As long as this limit is not exceeded, 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. The same test is applied when the interval timer interrupts an internal process.

This 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 pushed into the background to give other processes a chance. This is admittedly a built-in strategy at the microlevel. Parent processes can in fact only control the allocation of computing time to their children in larger portions (on the order of seconds) by means of the procedures *start* and *stop internal process*.

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 time quantums used by it; these quantities are denoted *start time* and *run time*.

7.3 Storage Allocation and Protection

An internal process can only create child processes within its own storage area. The monitor does not check whether storage areas of child processes overlap each other. This freedom can be used to implement time-sharing of a common storage area among several processes as described in Sections 10.2 and 10.4.

During creation of an internal process the parent must specify the values of the *protection register* and the *protection key* used by the child. In the protection register each bit corresponds to one of the eight possible protection keys; if a bit is zero the process can change or execute storage words with the corresponding key.

The protection key is the key that is set in all storage words of the child process itself. A parent process can only allocate a subset of its own protection keys to a child. It has complete freedom to allocate identical or different keys to its children. The keys remain accessible to the parent after creation of a child.

7.4 Message Buffers and Process Descriptions

The monitor only has room for a finite number of message buffers and tables describing internal processes and the so-called area processes (files on the backing store used as external processes). A message buffer is selected when a message is sent to another process; it is released when the sending process

receives an answer. A process description is selected when an internal process creates another internal process or an area process, and released when the process is removed.

Thus it is clear that message buffers and process descriptions only assume an identity when they are actually used. As long as they are unused, they can be regarded as anonymous pools of resources. Consequently it is sufficient to specify the maximum number of each resource an internal process can use. These so-called *buffer claim*, *internal claim*, and *area claim* 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 an internal process can exchange simultaneously with other internal and external processes. The internal claim limits the number of children an internal process can have at the same time. The area claim defines how many backing store areas an internal process can 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 again. Thus at any moment the claims define the number of resources that can still be used by the process.

7.5 Peripheral Devices

A distinction has been made between peripheral devices and external processes. An external process is created when a name is assigned to a device.

Thus it is also true of peripheral devices that they only assume an identity when they are actually used for input/output. Indeed the whole idea of identification by name is to give the operator complete freedom in allocation of devices. 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 peripherals, 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 of configuration must be defined in terms of physical devices and not in terms of external processes, since a parent generally speaking does not know in advance which documents its children are going to use.

Configuration control is exercised as follows. From the point of view 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 machine word. The process descriptions of peripheral devices include a word in which each bit indicates whether the corresponding internal process is a *potential user* of the device. Another word indicates the *current user* that has reserved the device in order to obtain exclusive access to a document.

Initially the basic operating system s is a potential user of all peripherals. A parent process can *include* or *exclude* a child as a user of any device, provided the parent is also a user of it:

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

During 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:

The device must be an external process with a unique name.

The internal process must be a user of the device.

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. Clearly there is a need to be able to prevent an internal process from reserving an excessive amount of space in the catalog or on the backing store as such. It seems difficult, however, to specify a reasonable rule in the form of a claim that is defined once and for all when a child process is created. The main difficulty is that catalog entries and data areas can survive the removal of the process that created them; in other words backing storage is a resource a parent process can loose permanently by allocating it to its children.

As a half-hearted solution we have introduced the concept of *privileged monitor procedures*. A parent process must supply 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 mask.

At present the privileged functions include all monitor procedures that:

change the catalog on the backing store,
create and remove names of peripheral devices,
change the real-time clock.

8 MONITOR FEATURES

This chapter is a survey of specific monitor features such as internal interruption, the real-time clock, conversational access from consoles, and permanent storage of files on the backing store. Although these are not essential primitive concepts, they are indispensable features of practical multiprogramming systems.

8.1 Internal Interruption

The monitor can assist internal processes with the detection of infrequent events such as violation of storage protection or arithmetic overflow. This causes an interruption of the internal process followed by a jump to an *interrupt procedure* within the process.

The interrupt procedure is defined by calling the monitor procedure:

set interrupt(interrupt address, interrupt mask)

When an internal interrupt occurs, the monitor stores the values of registers at the head of the interrupt procedure and continues execution of the internal process in the body of the procedure:

interrupt address:	working registers
	instruction counter
	interrupt cause
	(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
breakpoint forced by parent

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*. The process does not receive any more computing time in this state, but from the point of view of other processes it is still an existing process. The parent of the erroneous process can, however, reactivate it by means of stop and start.

A parent can force a *breakpoint* 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 continues in the interrupt procedure; fourth, start the child again.

8.2 Real-Time Clock

Real time is measured by means of a hardware interval timer, which counts modulo 16384 in units of 0.1 msec and interrupts the computer regularly (normally every 25.6 msec).

The monitor uses this 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 let the clock express the time interval elapsed since midnight 31 December 1967 in units of 0.1 msec.

The interval timer is also used to implement an external process that permits the synchronization of internal processes with real time. All internal processes can send messages to this *clock process*. After the elapse of a time interval specified in the message, the clock process returns an answer to the sender. In order to avoid a heavy overhead time of clock administration, the clock process only examines its queue every second.

8.3 Console Communication

A multiprogramming system encourages a conversational mode of operation, in which users interact directly with internal processes from typewriter

consoles. The external processes for consoles clearly reflect this objective.

Initially all program execution is ordered by human operators who communicate with the basic operating system. It would be very wasteful if the operating system had to examine all consoles regularly for possible operator requests. Therefore our first requirement is that consoles be able to activate internal processes by sending messages to them. Note that other external processes are only able to receive messages.

Second, it must of course be possible for an internal process to open a conversation with any console.

Third, a console should accept messages simultaneously from several internal processes. This will enable us to control more than one internal process from the same console, which is valuable in a small installation.

In short, consoles should be independent processes that can open conversations with any internal process and vice versa. The console should assist the operator with the identification of the internal processes using it.

An operator opens a conversation by depressing an interrupt key on the console. This causes the monitor to select a line buffer and connect it to the console. The operator must now identify the internal process to which his message is addressed. Following this he can input a message of one line, which is delivered in the queue of the receiving process.

A message to the basic operating system s can, for example, look like this (the word in italics is output by the console process in response to the key interrupt):

to s
new pbh run

An internal process opens a conversation with a console by sending a message to it. Before the input/output operation is initiated, the console identifies the internal process to the operator. This identification is suppressed after the first of a series of messages from the same process.

In the following example internal processes a and b share the same console for input/output. Process identifications are in italics:

to a
first input line to a
second input line to a
from b
first output line from b
second output line from b
from a
first output line from a
etc.

Note that these processes are unaware of their sharing the same console. From the point of view of internal processes the identification of user processes makes it irrelevant whether the system contains one or more consoles. (Of course one cannot expect operators to feel the same way about it).

8.4 Files on Backing Store

8.4.1 Introduction

The monitor permits semi-permanent storage of files on a backing store consisting of one or more drums and disks. The monitor makes these appear as a single backing store with a number of segments of 256 words each. This *logical backing store* is organized as a collection of named *data areas*. Each area occupies a consecutive number of segments on a single backing store device. A fixed part of the backing store is reserved for a *catalog* describing the names and locations of data areas.

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

8.4.2 Catalog Entries

The catalog is a fixed area on the backing store divided into a number of *entries* identified by unique *names*. Each entry is of fixed length and consists of a *head*, which identifies the entry, and a *tail*, which contains the rest of the information. The monitor distinguishes between entries describing data areas on the backing store and entries describing other things.

An entry is *created* by calling the monitor procedure:

create entry(name, tail, result)

The first word of the *tail* defines the *size* of an area to be reserved and described in the entry; if the size is negative or zero, no area is reserved. The rest of the tail contains nine *optional parameters*, which can be selected freely by the internal process.

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)

The catalog 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 only require the input of a single segment unless the catalog is filled to the brim. The allocation of data areas is speeded up by keeping a bit table of available segments within the monitor. In practice the creation or modification of an entry therefore requires only the input and output of a single catalog segment.

8.4.3 Catalog Protection

Since many users share the backing store as a common data base, it is vital that they have a means of protecting their files gainst unintentional modification or complete removal. The protection system used is similar to the storage protection system: each catalog entry is supplied with a *catalog key* in its head; the rules of access within an internal process are defined by a *catalog mask* set by the parent of the internal process. Each bit in this mask corresponds to one of 24 possible catalog keys; if a bit is one, the internal process can modify or remove entries with the corresponding key; otherwise it can only look up these entries. A parent can only allocate a subset of its own catalog keys to a child process. Initially the basic operating system owns all keys.

In order to prevent the catalog and the rest of the backing store from being filled with irrelevant data, the concept of *temporary entry* is introduced.

This is an entry that can be removed by another internal process as soon as the internal process that created the entry has been removed. Typical examples are working areas used during program compilation and data areas created, but not removed, by faulty programs.

This concept is implemented as follows. After creation of an internal process, the monitor increases an integer *creation number* by one and stores it within the new process description. Each time an internal process creates a catalog entry, the monitor includes its creation number in the entry head indicating that it is temporary. Internal processes can at any time scan the catalog and remove all temporary entries provided the corresponding creators no longer exist within the monitor. Thus in accordance with our basic philosophy the monitor only provides the necessary mechanism for the handling of temporary entries, but leaves the actual strategy of removal to the hierarchy of processes.

In order to ensure the survival of a catalog entry, an internal process must call the privileged monitor function:

permanent entry(name, catalog key, result)

to replace the creation number with a catalog key. A process can of course only set one of its own keys in the catalog; otherwise it might fill the catalog with highly protected entries, which could be difficult to detect and remove.

8.4.3 Area Processes

In order to be used for input/output a data area must be looked up in the catalog and described as an 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 parent processes (Section 7.4).

The backing store is a random access medium that serves as a common data base. ln order to utilize this property fully internal processes should be able to input simultaneously from the same area (e.g. when several copies of the Algol compiler are executed in parallel). On the other hand access to an area should be exclusive during output, because its content is undefined from the point of view of other processes.

Consequently we distinguish between internal processes that are *potential users* of an area process and the single process that may have *reserved* the area exclusively. This distinction was also made for peripheral devices (Section 5.2), but the rules of access are different here: An internal process is a user of an area after the creation of it. This enables the internal process to perform input as long as no other process reserves it. An internal process can reserve an area process if its catalog mask permits modification of the corresponding catalog entry. After reservation the internal 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, for instance, during the detection and removal of temporary entries. Only the monitor itself, however, can perform output to the catalog.

9 SYSTEM IMPLEMENTATION

This chapter gives important details about the implementation as well as figures about the size and performance of the system.

9.1 Interruptable Monitor Functions

Some of the monitor functions are too long to be executed entirely in the disabled mode, e.g. updating of the catalog on the backing store and creation, start, stop, and removal of processes. These so-called *process functions* are called as other monitor procedures, but behind the scenes they are executed by an anonymous internal process, which only operates in disabled mode for short intervals while updating monitor tables, otherwise the anonymous process shares computing time 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 the process description is linked to the event queue of the anonymous process that is activated. The anonymous process serves the calling processes one by one and returns them to the time slice queue after completion of each function.

Process functions are interruptable like other internal processes. From the point of view of calling processes, however, process functions are indivisible, since (1) they are executed only by the anonymous process one at a time in their order of request, and (2) calling processes are delayed until the functions are completed.

The following monitor procedures are implemented as interruptable functions:

create entry
look up entry
change entry
rename entry
remove entry
permanent entry
create area process
create peripheral process
create internal process
start internal process
stop internal process
modify internal process
remove process

9.2 Stopping Processes

According to theory an internal process cannot be stopped while input/output is in progress within its storage area (Section 6.3). This requirement is inevitable in the case of high-speed devices such as a drum or a magnetic tape station, which are beyond program control during input/output. On the other hand it is not strictly necessary to enforce this for low-speed devices controlled by the monitor on a character-by-character basis.

In practice the monitor handles the stop situation as follows:

Before an external process initiates *high-speed input/output*, it examines the state of the sending process. If the sender is stopped (or waiting to be stopped), input/output is not initiated, but the external process immediately returns an answer with block length zero; the sender must then repeat

input/output after restart. If the sender is not stopped, its stop count is increased and input/output is initiated. Note that if the stop count was increased immediately after the sending of a message, the sending process could only be stopped after completion of all previous operations pending in the external queue. By increasing the stop count as late as possible, we ensure that high-speed peripherals at most prevent the stopping of internal processes during a single block transfer.

Low-speed devices never increase the stop count. During output an external process fetches one word at a time from the sending process and outputs it character by character regardless of whether the sender is stopped meanwhile. Before fetching a word the external process examines the state of the sender. If it is stopped (or waiting to be stopped), output is terminated by an answer defining the actual number of characters output; otherwise output continues. During input an external process examines the state of the sender after each character. If the sender is stopped (or waiting to be stopped), input is terminated by an answer; otherwise the character is stored and input continues. Some devices, such as the typewriter, lose the last input character when stopped; others, such as the paper tape reader, do not. It can be seen that low-speed devices never delay the stopping of a process.

9.3 System Size

After initial system loading the monitor and the basic operating system s occupy a fixed part of the internal store. The size of a typical system is as follows:

		words:
monitor procedures:		2400
code for external processes:		1150
clock	50	
backing store	100	
typewriters	300	
paper tape readers	250	
paper tape punches	150	
line printers	100	
magnetic tape stations	200	
process descriptions and buffers:		1250
15 peripheral devices	350	
20 area processes	200	
6 internal processes	200	
25 message buffers	300	
6 console buffers	200	
basic operating system s		1400
total system		6200

It should be noted that the 6 internal processes include the anonymous process and the basic operating system, thus leaving room for 4 user processes. As a minimum the standard programs (editor, assembler, and compilers) require an internal process of 5–6000 words for their execution. This means that a 16 k store can only hold the system plus 1–2 standard programs, while a 32 k store enables parallel execution of 4 such programs. A small store can of course hold more programs, if these are written in machine code and executed without the assistance of standard programs.

9.4 System Performance

The following execution times of monitor procedures are conservative estimates based on a manual count of instructions. The reader should keep in mind that the basic instruction execution time of the RC 4000 computer is 4 μsec. A complete conversation between two internal processes takes about 2 milliseconds distributed as follows:

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

It can be seen that one internal process can activate another internal process in 0.6 msec, this is also approximately the time required to activate an external process. An analysis shows that the 2 msec required by an internal communication are used as follows:

	percent
validity checking	25
process activation	45
message buffering	30

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

The maximum time spent in the disabled mode within the monitor limits the system's response to real-time events. The monitor procedures themselves are only disabled for 0.2–1 msec. The situation is worse in the case of interrupt procedures that handle low-speed devices with hardware buffers, because the monitor empties or fills such buffers in the disabled mode after each interrupt. For the paper tape reader (flexowriter input) and the line printer, the worst-case figures are:

empty reader buffer (256 characters)	20 msec
fill printer buffer (170 characters)	7 msec

It should be noted, however, that these buffers normally only contain 64–70 characters corresponding to 4–5 msec. The worst-case situations can be remedied either by using smaller input/output areas within internal processes, or by replacing these external processes with dedicated internal processes (Section 5.4).

Finally we shall look at the interruptable monitor functions. An internal process of 5000 words can be created and controlled by a parent process with the following speed:

	msec
create internal process	3
modify internal process	2
start internal process	26
stop internal process	4
remove internal process	30

Most of the time required to start and remove an internal process is used to set storage protections keys.

Assuming that the backing store is a drum with a transfer time of 15 msec per segment, the catalog can be accessed with the following speed:

	msec
create entry	38
look up entry	20
change entry	38
rename entry	85
remove entry	38
permanent entry	38

The execution time of process functions should be taken with some reservations. First it must be remembered that process functions, like other internal processes, can be delayed for some time before they receive a time slice. In practice process functions will be activated immediately as long as they have not used a complete time slice (Section 7.2). Second one must take into consideration the fact that process function calls are queued within the monitor. Thus when a process wants to stop another process, the worst thing that can happen is that the anonymous process is engaged in updating the catalog. In this situation the stop is not initiated before the catalog has been updated. One also has to keep in mind that process functions share the drum or disk with other processes, and must wait for the completion of all input/output operations that preceed their own in the drum or disk queue. The execution times given here assume that process functions and catalog input/output are initiated instantly.

9.5 System Tape

The first version of the multiprogramming system consists of the monitor, the basic operating system s, and a program for initializing the catalog. It is programmed in the Slang 3 language. Before assembly the system is edited to include process descriptions of the peripheral devices connected to a particular installation and to define the following *options*:

number of storage bytes
number of internal processes
number of area processes
number of message buffers
number of console buffers
maximum time slice
inclusion of code for external processes
backing store configuration
size of catalog

The system is delivered in the form of a binary paper tape, which can autoload and initialize itself. After loading the system starts the basic operating system. Initially the operating system executes a program that can initialize the backing store with catalog entries and binary Slang programs input from paper tape. When this has been done, the operating system is ready to accept operator commands from consoles.

10 SYSTEM POSSIBILITIES

The strength of the monitor is the generality of its basic concepts, its weakness that it must be supported by operating systems to obtain realistic multiprogramming. We believe that the ultimate limits to the use of the system will depend on the imagination of designers of future operating systems. The purpose of this chapter is to stimulate creative thinking by pointing out a few of the possibilities inherent in the system.

10.1 Identification of Documents

In tape-oriented installations, operating systems should assist the operator with automatic identification of magnetic tapes. At present the external process concept gives the operator complete freedom to mount a magnetic tape on any station and identify it by name. When a tape station is set in the *local* mode, the monitor immediately removes its name to indicate that the operator has interfered with it. The station gives an interrupt when the operator returns it to the *remote* mode. Thus the monitor distinguishes between three states of a tape station:

document removed (after intervention)
unidentified document mounted (after remote interruption)
identified document mounted (after process creation)

It is a simple matter to introduce a *watch-dog process* in the monitor, to which internal processes can send messages in order to receive answers each time an unidentified tape is mounted somewhere. After reception of an answer, an internal process can give the actual station a temporary name, identify the tape by reading its label, and rename it accordingly.

Automatic identification requires general aggreement on the format of tape labels, at least to the extent of assigning a standard position to the names of tapes.

10.2 Temporary Removal of Programs

We have not imposed any restrictions on individual programs with respect to their demand for storage, run time, and peripherals. It is taken for granted that some programs will need most of the system resources for several hours. Such large programs must not, however, prevent other users from obtaining immediate access to the machine in order to execute more urgent programs of short duration. Thus the system must permit temporary removal of a program in order to make its storage area and peripherals available for other programs. One example, where this is absolutely necessary, is the periodic supervision of a real-time process combined with the execution of large background programs in idle intervals.

A program can be removed temporarily by stopping the corresponding internal process and dumping its storage area on the backing store by an output operation. Note that this dump automatically includes all children and descendants created within the area. The monitor is only aware of the process being stopped; it is still described within the monitor and can receive messages from other processes.

It is now possible to create and start other processes in the same storage area, since the monitor does not check whether internal processes overlap each other as long as they remain within their parent processes. Peripherals can also be taken from the dumped process and assigned to others simply by mounting new documents and renaming the peripherals.

Temporary removal makes sense only if it is possible to restart a program at a later stage. This requires reloading the program into its original storage area as well as mounting and repositioning of its documents. After restart the internal process can detect interference with its documents in one of two ways: either it finds that a document does not exist any more, whereupon it must ask the operator to mount and name it; or it discovers that an existing document no longer is reserved by it, meaning that the operator has mounted

it, but that it needs to be repositioned. These cases are indicated by the result parameter after a call of *wait answer*.

The need for repositioning can also arise during normal program execution, if the operator interferes with a peripheral device (by mistake or in order to move a document to a more reliable device). Consequently all major programs should consider each input/output operation as a potential restart situation.

10.3 Batch Processing

In the design of a batch processing system the distinction between parent and child processes prevents the batch of programs from destroying the operating system. Note that in general an operating system must remove a child process (and not merely stop it) to ensure that all its resources are released again (Section 7.4). Even then, it must be remembered that messages sent by a child to other processes remain in their queues until these processes either answer them or are removed (Section 4.4).

The multiprogramming capabilities can be utilized to accept job requests in a conversational mode during execution of the batch. Thus a *batch processing* system can include facilities for *remote job entry* combined with *priority scheduling* of programs.

10.4 Time-Sharing

The basic requirement of a *time-sharing* system, in which a large number of users have conversational access to the system from consoles, is the ability to swap programs between the internal store and the backing store. A time-sharing operating system must create an internal process for each user, and make these processes share the same storage area by frequent removal and restart of programs (say, every few seconds). The problem is that stopping a process temporarily also means stopping its communication with peripherals. Thus in order to keep typewriter input/output alive while a user process is dumped, the system must include an internal process that buffers all data between programs and consoles.

10.5 Real-Time Scheduling

We conclude these hints with an example of a *real-time* system. The application we have in mind is a process control system, in which a number of

programs must perform data logging, alarm scanning, trend logging, and so forth periodically under the real-time control of an operating system.

This can be organized as follows: initially all task programs send messages to the operating system and wait for answers. The operating system communicates with the clock process and is activated every second in order to scan a time table of programs. If the real time exceeds the start time of a task program, the operating system activates the program by an answer. After completion of its task, the program again sends a message to the operating system and waits for the answer. In response the operating system increases the start time of the program by the period between two successive executions of the task.

Acknowledgements

The design of the system is based on the ideas of Jørn Jensen, Søren Lauesen, and the author; Leif Svalgaard participated in its implementation.