The Nucleus of a Multiprogramming System*

(1970)

This paper describes the philosophy and structure of a multiprogramming system that can be extended with a hierarchy of operating systems to suit diverse requirements of program scheduling and resource allocation. The system nucleus simulates an environment in which program execution and input/output are handled uniformly as parallel, cooperating processes. A fundamental set of primitives allows the dynamic creation and control of a hierarchy of processes as well as the communication among them.

1 Introduction

The multiprogramming system developed by Regnecentralen for the RC 4000 computer is a general tool for the design of operating systems. It allows the dynamic creation of a hierarchy of processes in which diverse strategies of program scheduling and resource allocation can be implemented.

For the designer of advanced information systems, a vital requirement of any operating system is that it allow 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 conversational access.

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,

*P. Brinch Hansen, The nucleus of a multiprogramming system, *Communications of the ACM 13*, 4 (April 1970), 238–242.

matter because 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.

In the following, the philosophy and structure of the RC 4000 multiprogramming system is explained. The discussion does not include details of implementation; size and performance are presented, however, to give an idea of the feasibility of this approach. The functional specifications of the multiprogramming system are described in detail in a report (Brinch Hansen 1969a) available from Regnecentralen.

2 System Nucleus

Our basic attitude during the designing was to make no assumptions about the particular strategy needed to optimize a given type of installation, but to concentrate on the fundamental aspects of the control of an environment consisting of parallel, cooperating processes.

Our first task was to assign a precise meaning to the process concept, i.e. to introduce an unambiguous terminology defining what a process is and how it is implemented on the actual computer.

The next step was to select primitives for the synchronization and transfer of information among parallel processes.

Our final decisions concerned the rules for the dynamic creation, control, and removal of processes.

The purpose of the system nucleus is to implement these fundamental concepts: simulation of processes; communication among processes; creation, control, and removal of processes.

3 Processes

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.

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.

In connection with input/output, the system 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, such as a deck of punched cards, a printer form, a reel of magnetic tape, or a file on the backing store.

An *external process* is the input/output of a given document identified by a unique process name. This concept implies that internal processes can refer to documents by name without knowing the actual devices on which they are mounted.

Multiprogramming and communication between internal and external processes is coordinated by the system nucleus—an interrupt response program with complete control of input/output, storage protection, and the interrupt system. We do not regard the system nucleus 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 implement our process concept and primitives that processes can invoke 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

In a system of parallel, cooperating processes, mechanisms must be provided for the synchronization of two processes during a transfer of information.

Dijkstra (1965) has demonstrated that indivisible lock and unlock operations operating on binary semaphores are sufficient primitives from a logical point of view. We have been forced to conclude, however, that the semaphore concept alone does not fulfill our requirements of safety and efficiency in a dynamic environment in which some processes may turn out to be black

sheep and break the rules of the game.

Instead we have introduced message buffering within the system nucleus as the basic means of process communication. The system nucleus administers a common pool of *message buffers* and a *message queue* for each process.

The following primitives 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 identity of the message buffer.

Wait message delays the requesting 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 identity of the message buffer. The buffer is removed from the queue and made 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 requesting 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 system nucleus in response to a message addressed to a nonexisting process.

The procedure wait message forces a process to serve its queue on a first-come, first-served basis. The system, however, also includes two primitives that enable a process to wait for the arrival of the next message or answer and serve its queue in any order.

This communication system has the following advantages.

The multiprogramming system is dynamic in the sense that processes 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 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 (i.e. by means of a message) they need a common identification of it in order to agree on when it is terminated (i.e. 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 occurrence of erroneous or malicious processes in the system (e.g. undebugged programs). This is tolerable only if the system nucleus ensures that no process can interfere with a conversation between two other processes. This is done by storing the identity of 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 queueing of buffers, which enables a sending process to continue immediately after delivery of a message or an answer, regardless of whether or not the receiver is ready to process it.

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 this case, the system nucleus leaves all messages from the removed process undisturbed in the queues of other processes. When these processes answer them, the system nucleus returns the buffers to the common pool.

The reverse situation is also possible: during the removal of a process, the system nucleus finds unanswered messages sent to the process. These are returned as dummy answers to the senders.

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 were 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. Consequently a limit is set 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.

5 External Processes

Originally the communication primitives were designed for the exchange of messages between internal processes. Later we also decided to use *send*

message and *wait answer* for communication between internal and external processes.

For each kind of external process, the system nucleus contains a piece of code that interprets a message from an internal process and initiates input/output using a storage area specified in the message. When input/output is terminated by an interrupt, the nucleus generates an answer to the internal process with information about actual block size and possible error conditions. This is essentially the implementation of the external process concept.

We consider it to be an important aspect of the system that internal and external processes are handled uniformly as independent, self-contained processes. The difference between them is merely a matter of processing capability. A consequence of this is that any external process can be replaced by an internal process of the same name if more complex criteria of access and response become desirable.

External processes are created on request from internal processes. *Creation* is simply the assignment of a name to a particular peripheral device. To guarantee internal processes exclusive access to sequential documents, primitives are available for the *reservation* and *release* of external processes.

Typewriter consoles are the only external processes that can send messages to internal processes. The operator opens a conversation by pushing an interrupt key and typing the name of the internal receiver followed by a line of text.

A file on the backing store can be used as an external process by copying a description of the file from a catalog on the backing store into the system nucleus; following this, internal processes can initiate input/output by sending messages to the file process.

Real-time synchronization of internal processes is obtained by sending messages to a clock process. After the elapse of a time interval specified in the message, the clock returns an answer to the sending process.

In general, external processes can be used to obtain synchronization between internal processes and any signal from the external world. For example, an internal process may send a message to a watchdog process and receive an answer when a magnetic tape is mounted on a station. In response, the internal process can give the station a temporary name, identify the tape by reading its label, and rename the station accordingly.

6 Internal Processes

A final set of primitives in the system nucleus allows the creation, control, and removal of internal processes.

Internal processes are created on request from other internal processes. *Creation* involves the assignment of a name to a contiguous storage area selected by the parent process. The storage area must be within the parent's own area.

After creation, the parent process can load a program into the child process and *start* it. The child process now shares computing time with other active processes including the parent process.

On request from a parent process, the system nucleus waits for the completion of all input/output initiated by a child process and *stops* it. In the stopped state, the process can still receive messages and answers in its queue. These can be served when the process is restarted.

Finally, a parent process can *remove* a child process in order to assign its storage area to other processes.

According to our philosophy, processes should have complete freedom to choose their own strategy of program scheduling. The system nucleus only supplies the essential primitives for initiation and control of processes. Consequently, the concepts of program loading and swapping are not part of the nucleus. Time-sharing of a common storage area among children on a swapping basis is possible, however, because the system does not check whether internal processes overlap each other as long as they remain within the storage areas of their parents. Swapping from process A to process B can be implemented in a parent process as follows: stop(A); output(A); input(B); start(B).

7 Process Hierarchy

The idea of the system nucleus 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 primitives 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 of operator communication, program scheduling, and resource allocation; but it is essential for the orderly growth of the system

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 system nucleus and a basic operation system, S, which can create parallel processes, A, B, C, etc., on request from consoles. The 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, and F. This is illustrated by Fig. 1, 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 of processes.

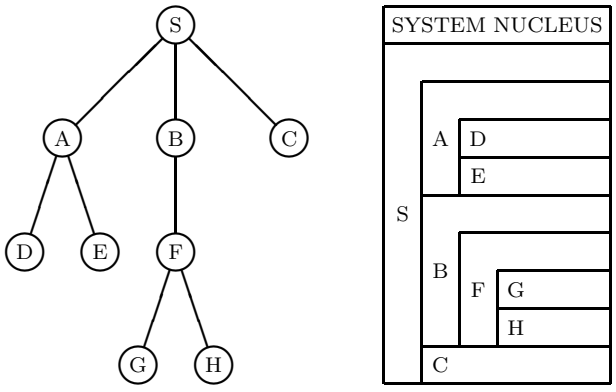

Figure 1 Process tree and storage allocation.

In this multiprogramming system, all privileged functions are implemented in the system nucleus, 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 children. The only rules enforced by the nucleus are the following: A process can only allocate a subset of its own resources (including storage and message buffers) to its children; a process can only start, stop, and remove its own children (including their descendants). After removal of a process, its resources are returned to the parent process. Initially all system resources are owned by

the basic operating system S. For details of process control and resource allocation, the reader should consult the manual of the system (Brinch Hansen 1969a).

We emphasize that the only function of the family tree is to define the rules of process control and resource allocation. Computing time is shared by round-robin scheduling among active processes regardless of their position in the hierarchy, and each process can communicate with all other processes.

Regarding the future development of operating systems, the most important characteristics of the system can now be seen as the following:

1. New operating systems can be implemented as other programs without modification of the system nucleus. In this connection, we should mention that the Algol and Fortran languages for the RC 4000 contain facilities for calling the nucleus 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, provided there is common agreement on the possible communication between parents and children.

8 Implementation

The RC 4000 is a 24-bit, binary computer with typical instruction execution times of 4 microseconds (Brinch Hansen 1969b). It permits practically unlimited expansion of the internal store and standardized connection of all kinds of peripherals. Multiprogramming is facilitated by program interruption, storage protection, and privileged instructions.

The present implementation of the system makes multiprogramming feasible with a minimum store of 16K–32K words backed by a fast drum or disk. The system nucleus includes external processes for a real-time clock, typewriters, paper tape input/output, line printer, magnetic tape, and files on the backing store. The size of the nucleus and the basic operating system is as follows:

	words
primitives	2400
code for external processes	1150
process descriptions and buffers	1250
system nucleus	4800
basic operating system	1400
	6200

The communication primitives are executed in the uninterruptable mode within the system nucleus. The execution times of these set a limit to the system's response to real-time events:

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

An analysis shows that the 2 milliseconds required by a complete conversation (the sum of the four primitives) are used as follows:

	percent
validity checking	25
process activation	45
message buffering	30

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

The primitives for creation, start, stop, and removal of processes are implemented in an anonymous internal process within the system nucleus to avoid intolerably long periods in the uninterruptable mode. Typical execution times for these are:

	msec
create process	3
start process	26
stop process	4
remove process	30

The excessive times for the start and removal of an internal process are due to the peculiar storage protection system of the RC 4000, which requires the setting of a protection key in every storage word of a process.

9 Conclusion

Ideas similar to those described here have been suggested by others (Harrison 1967; Huxtable 1967; Wichmann 1968). We have presented our system because we feel that, taken as a whole, it represents a systematic and practical approach to the design of replaceable operating systems. As an inspiration to other designers, it is perhaps most important that it illustrates a sequence of design steps leading to a general nucleus, namely, the definition of the process concept, the communication scheme, and the dynamic creation and structuring of processes.

We realize, of course, that a final evaluation of the system can only be made after it has been used to design a number of operating systems.

Acknowledgements

The design philosophy was developed by Jørn Jensen, Søren Lauesen, and the author. Leif Svalgaard participated in the implementation and testing of the final product.

Regarding fundamentals, we have benefited greatly from Dijkstra's analysis of cooperating sequential processes.

References

Brinch Hansen, P., Ed. 1969a. *RC 4000 Software: Multiprogramming System*. Regnecentralen, Copenhagen, Denmark, (April).

Brinch Hansen, P. Ed. 1969b. *RC 4000 Computer: Reference Manual*. Regnecentralen, Copenhagen, Denmark, (June).

Dijkstra, E.W. 1965. Cooperating sequential processes. Technological University, Eindhoven, The Netherlands, (September).

Harrison, M.C., and Schwartz, J.T. 1967. SHARER, a time sharing system for the CDC 6600. *Communications of the ACM 10*, 10 (October), 659–665.

Huxtable, D.H.R., and Warwick, M.T. 1967. Dynamic supervisors—their design and construction. *ACM Symposium on Operating System Principles*, Gatlinburg, TN, (October).

Wichmann, B.A. 1968. A modular operating system. *Proceedings of the IFIP Congress 68*, Edinburgh, United Kingdom. North Holland, Amsterdam, The Netherlands, 1969, 548–556.