AN OUTLINE OF A COURSE ON OPERATING SYSTEM PRINCIPLES

PER BRINCH HANSEN

(1971)

In 1970 the author began writing a comprehensive textbook on operating system principles. This is a description of its structure and how far it had progressed a year later.

COMPUTER SCIENCE AND OPERATING SYSTEMS

In November 1970 I began writing a textbook on operating system principles at Carnegie-Mellon University. This is a description of its structure and how far it has progressed.

The goal is to give students of computer science and professional programmers a general understanding of operating systems. The only background required is an understanding of the basic structure of computers and programming languages and some practical experience in writing and testing non-trivial programs. In a few cases a knowledge of elementary calculus and probability theory is also needed. The components of the course are well-known to a small group of designers, but most operating systems reveal an inadequate understanding of them.

The first and most obvious problem is to delimit the subject and consider its place in computer science education. I define an *operating system* as a set of manual and automatic procedures which enable a group of users to share a computer system efficiently. The keyword in this definition is *sharing*: it means competetion for the use of physical resources but also cooperation

P. Brinch Hansen, An outline of a course on operating system principles. In *Operating Systems Techniques*, Proceedings of a Seminar at Queen's University, Belfast, Northern Ireland, August–September 1971. C. A. R. Hoare and R. H. Perrott, Eds. Academic Press, New York (1972), 29–36. Copyright © 1972, Academic Press.

among users exchanging programs and data on the same computer system. All shared computer systems must *schedule* user computations in some order, *protect* them against one each other, and give them means of *long-term storage* of programs and data. They must also perform *accounting* of the cost of computing and *measure* the actual performance of the system.

In early computer systems, operators carried out most of these functions, but during the last fifteen years the programs that we call operating systems have gradually taken over these aspects of sharing.

Although most components of present computers are sequential in nature, they can work simultaneously to some extent. This influences the design of operating systems so much that the subject can best be described as the *management of shared multiprogramming systems*.

Operating systems are large programs developed and used by a changing group of people. They are often modified considerably during their lifetime. Operating systems must necessarily impose certain restrictions on all users. But this should not lead us to regard them as being radically different from other programs. They are just examples of large programs based on fundamental principles of computer science. The proper aim of education is to identify these fundamentals.

The student should realize that principles and methods of resource sharing have a general utility that goes beyond operating systems. Any large programming effort will be heavily influenced by the presence of several levels of storage, by the possibility of executing smaller tasks independently, and by the need for sharing a common set of data among such tasks. We find it convenient to distinguish between operating systems and user computations because the former can *enforce* certain rules of behavior on the latter. It is important, however, to realize that each level of programming solves some aspect of resource allocation.

I argue therefore that the study of operating systems leads to the recognition of general principles which should be taught as part of a core of computer science. Assuming that the student has an elementary background in *programming languages*, *data structures* and *computer organization*, the course concentrates on the following areas of computer science: *concurrent computations*, *resource sharing* and *program construction*.

Let us look at the course in some detail. It consists of eight parts which are summarized in the Appendix. The following is a more informal presentation of its basic attitude.

TECHNOLOGICAL BACKGROUND

The necessity of controlling access to shared computer systems automatically is made clear by simple arguments about the poor utilization of equipment in an *open shop* operated by the users themselves, one at a time. As a first step in this direction, I describe the classical *batch processing system* which carries out computations on a main computer while a smaller computer prepares and prints magnetic tapes. The strict sequential nature of the processors and their backing storage in this early scheme made it necessary to prevent human interaction with computations and schedule them in their order of arrival inside a batch.

These restrictions on scheduling disappear to some extent with the introduction of multiprogramming techniques and large backing stores with random access. This is illustrated by two simple operating systems: the first one is a *spooling system* which handles a continuous stream of input, computation and output on a multiprogrammed computer with drum storage; the other is an *interactive system* in which main storage is shared cyclically among several computations requested from remote terminals.

Through a chain of simple arguments the student gradually learns to appreciate the influence of *technological constraints* on the service offered by operating systems.

THE SIMILARITY OF OPERATING SYSTEMS

The main theme of the course is the similarity of problems faced by all operating systems. To mention one example: all shared computer systems must handle concurrent activities at some level. Even if a system only schedules one computation at a time, users can still make their requests simultaneously. This problem can, of course, be solved by the users themselves (forming a waiting line) and by the operators (writing down requests on paper). But the observation is important, since our goal is to handle the problems of sharing automatically.

It is also instructive to compare a batch processing and a spooling system. Both achieve high efficiency by means of concurrent activities: in a batch processing system independent processors work together; in a spooling system a single processor switches among independent programs. Both systems use backing storage (tape and drum) as a buffer to compensate for speed variations between the producers and consumers of data.

As another example, consider real-time systems for process control or conversational interaction. In these systems, concurrent processes must be able to exchange data in order to cooperate on common tasks. But again, this problem exists in all shared computer systems: in a spooling system user computations exchange data with concurrent input/output processes; and in a batch processing system we have another set of concurrent processes which exchange data by means of tapes mounted by operators.

So I find that all operating systems face a common set of problems. To recognize these we must reject the established classification of operating systems into batch processing, time sharing, and real time systems which stresses the dissimilarities of various forms of technology and user service. This does not mean that the problems of adjusting an operating system to the constraints of a certain environment are irrelevant. But the students will solve them much better when they have grasped the underlying common principles.

You will also look in vain for chapters on input/output and filing systems. For a particular operating system considerations about how these problems are handled are highly relevant; but again I have concentrated on the more elementary problems involved in these complicated tasks, namely, process synchronization, storage management and resource protection.

SEQUENTIAL AND CONCURRENT COMPUTATIONS

After this introduction, the nature of computations is described. A *computation* is a set of operations applied to a set of data in order to solve a problem. The operations must be carried out in a certain order to ensure that the results of some of them can be used by others. In a *sequential process* operations are carried out strictly one at a time. But most of our computational problems only require a partial ordering of operations in time: some operations must be carried out before others, but many of them can be carried out concurrently.

The main obstacles to the utilization of concurrency in computer systems are economy and human imagination. Sequential processes can be carried out cheaply by repeated use of simple equipment; concurrent computations require duplicated equipment and time-consuming synchronization of operations. Human beings find it extremely difficult to comprehend the combined effect of a large number of activities which evolve simultaneously with independent rates. In contrast, our understanding of a sequential process is independent of its actual speed of execution. All that matters is that operations are carried out one at a time with finite speed, and that certain relations hold between the data before and after each operation.

So sequential processes closely mirror our thinking habits, but a computer system is utilized better when its various parts operate concurrently. As a compromise, we try to partition our problems into a moderate number of sequential activities which can be programmed separately and then combined for concurrent execution. These processes are *loosely connected* in the sense that they can proceed simultaneously with arbitrary rates except for short intervals when they exchange data.

After a brief review of methods of structuring data and sequential programs, I consider the synchronizing requirements of *concurrent processes.* It is shown that the results of concurrent processes which share data cannot be predicted unless some operations exclude each other in time. Operations which have this property are called *critical regions.* Mutual exclusion can be controlled by a data structure, called a *semaphore*, consisting of a boolean, defining whether any process is inside its critical region, and a queue, containing the set of processes waiting to enter their regions.

A critical region is one example of a timing constraint or *synchronization* imposed on concurrent processes. Synchronization is also needed when some processes produce data which are consumed by other processes. The simplest *input/output relationship* is the exchange of *timing signals* between processes. The constraint here is that signals cannot be received faster than they are sent. This relationship can be represented by an integer semaphore accessed by *signal* and *wait* operations only.

Realistic *communication* between processes requires the exchange of data structures. This problem can be solved by synchronizing primitives operating on semaphores and data structures which are accessible to all the processes involved. It is tempting to conclude that critical regions, common data, and wait and signal operations are the proper concepts to include in a programming language. Experience shows that the slightest mistake in the use of these tools can result in erroneous programs which are practically impossible to correct because their behavior is influenced by external factors in a time-dependent, irreproducible manner.

A more adequate solution is to include *message buffers* as primitive data structures in the programming language and make them accessible only through well-defined *send* and *receive* operations. The crucial point of this language feature is that storage containing shared data (messages) is accessible to at most one process at a time. It has been proved that when a set of smaller systems with time-independent behavior are connected by means of message buffers only, the resulting system can also be made time-independent in behavior.

The most general form of process interaction is one in which a process must be delayed until another process has ensured that certain relationships hold between the components of a shared data structure. This form of synchronization can be expressed directly by means of *conditional critical regions.*

The conceptual simplicity of simple and conditional critical regions is achieved by ignoring the sequence in which waiting processes enter these regions. This abstraction is unrealistic for heavily used resources. In such cases, the operating system must be able to identify competing processes and control the scheduling of resources among them. This can be done by means of a *monitor*—a set of shared procedures which can delay and activate individual processes and perform operations on shared data.

Finally, I consider the problems of *deadlocks* and their prevention by hierarchical ordering of process interactions.

RESOURCE MANAGEMENT

Most of the previous concepts are now widely used. Far more controversial are the problems of how abstract computations are represented and managed on physical systems with limited resources. At first sight, problems caused by the physical constraints of computers seem to be of secondary importance to the computational problems we are trying to solve. But in practice most programming efforts are dominated by technological problems and will continue to be so. It will always be economically attractive to share resources among competing computations, use several levels of storage, and accept occasional hardware malfunction.

It seems unrealistic to look for a unifying view of how different kinds of technology are used efficiently. The student should realize that these issues can only be understood in economic terms. What we can hope to do is to describe the circumstances under which certain techniques will work well.

The implementation of the process concept is considered in two chapters on *processor multiplexing* and *storage organization.* The first of these describes the representation of processes and scheduling queues at the lowest level of programming and the implementation of synchronizing primitives. Hardware registers, clocks and interrupts are treated as technological tools which in many cases can be replaced by more appropriate concepts at higher levels of programming. The second of these chapters discusses the compromises between associative and location-dependent addressing, and the dynamic allocation of fixed and variable-length data structures in storage with one or more levels.

Following this, I discuss the influence of various *scheduling algorithms*: first-come first-served, shortest job next, highest response ratio next, round robin, and so on, on the behavior of the system in terms of average response times to user requests.

A CASE STUDY

At the end of the course, the conceptual framework is used to describe an existing operating system in depth using a consistent terminology.

I have selected the RC 4000 multiprogramming system (Brinch Hansen 1970) as a case study, because it is the only one I know in detail, and is a small, consistent design which illustrates essential ideas of concurrent processes, message communication, scheduling and resource protection.

THE CHOICE OF A DESCRIPTION LANGUAGE

So far nearly all operating systems have been written partly or completely in machine language. This makes them unnecessarily difficult to understand, test and modify. I believe it is desirable and possible to write efficient operating systems almost entirely in a *high-level language*. This language must permit *hierarchal structuring* of data and program, extensive *error checking* at compile time, and production of *efficient machine code*.

To support this belief, I have used the programming language *Pascal* (Wirth 1971) throughout the text to define operating system concepts concisely by algorithms. Pascal combines the clarity needed for teaching with the efficiency required for design. It is easily understood by programmers familiar with Algol 60 or Fortran, but is a far more natural tool than these for the description of operating systems because of the presence of data structures of type record, class and pointer.

At the moment, Pascal is designed for sequential programming only, but I extend it with a suitable notation for multiprogramming and resource sharing. I have illustrated the description of operating systems in Pascal elsewhere (Brinch Hansen 1971a, 1971b).

STATUS OF THE COURSE

I conceived the plan for the course in March 1970 and started to work on it in November 1970. Now, in November 1971, drafts have been written of parts 1–4, and 6 (see the Appendix). Most of the work on parts 5, and 7–8 remains to be done. It is unlikely that the structure of the course will change significantly, although the details certainly will.

APPENDIX: THE CONTENTS OF THE COURSE

1. An overview of operating systems

The purpose of an operating system. Technological background: manual scheduling, non-interactive scheduling with sequential and random access backing storage, interactive scheduling. The similarity of operating systems. Special versus general-purpose systems.

2. Sequential processes

Abstraction and structure. Data and operations. Sequential and concurrent computations. Methods of structuring data and sequential programs. Hierarchal program construction. Programming levels viewed as virtual machines. Our understanding and verification of programs.

3. Concurrent processes

Time-dependent programming errors in concurrent computations. Definition of functional behavior in terms of input/output histories. The construction of functional systems from smaller functional components. Concurrent systems with inherent time-dependent behavior: priority scheduling and shared processes.

Disjoint and interacting processes. Mutual exclusion of operations on shared data. Simple and conditional critical regions. Process communication by semaphores and message buffers. Explicit control of process scheduling by monitors.

The deadlock problem. Prevention of deadlocks by hierarchal ordering of process interactions.

4. Processor multiplexing

Short-term and medium-term scheduling. A computer system with identical processors connected to a single store. Peripheral versus central processors.

Process descriptions, states and queues. Processor execution cycle. Scheduling of critical regions by means of a storage arbiter. Implementation of the scheduling primitives wait, signal, initiate and terminate process. Influence of critical regions on preemption. Processor multiplexing with static and dynamic priorities. Implementation details: hardware registers, clock, interrupts. Timing constraints.

5. Storage organization

Properties of abstract and physical storage. Methods of address mapping: searching, key transformation and base registers.

Single-level storage: fixed partitioning, dynamic allocation of fixed and variable-length data structures. Compacting and fragmentation.

Hierarchal storage: swapping, demand paging and extended storage. Locality principle. Prevention of thrashing. Placement and replacement strategies. Hardware support.

Influence of input/output, process communication, and scheduling on storage allocation.

6. Scheduling algorithms

Objectives of scheduling policies. Queueing models of user requests and computations. Performance measures. A conservation law for a class of priority scheduling algorithms.

Non-preemptive scheduling: fixed priorities, first-come first-served, shortest job next, and highest response ratio next.

Preemptive scheduling: round robin with swapping. Methods of reducing transfers between storage levels. Scheduling with performance feedback.

7. Resource protection

The concept of a process environment of shared objects. Requirements of naming and protection. Existing protection mechanisms: privileged execution state, storage protection, file systems with private and public data, user password identification, protection levels and process hierarchies.

8. A case study

A detailed analysis of the structure, size and performance of the RC 4000 multiprogramming system.

Acknowledgements

Without the encouragement of Alan Perlis this work would not have been undertaken. I am indebted to Nico Habermann, Anita Jones and Bill Wulf who read and criticized all or part of the manuscript. I learned much from discussions with Tony Hoare. It should also be mentioned that without the foundation of laid by Edsger Dijkstra (1965) we would still be unable to separate principles from their applications in operating systems. The idea of looking upon the management of shared computer systems as a general data processing problem was inspired by a similar attitude of Peter Naur (1966) towards program translation.

References

Brinch Hansen, P. 1970. The nucleus of a multiprogramming system. *Communications of the ACM 13*, 4 (April), 238–250.

Brinch Hansen, P. 1971a. Short-term scheduling in multiprogramming systems. *3rd ACM Symposium on Operating System Principles*, Stanford University, Stanford, CA, (October), 101–105.

Brinch Hansen, P. 1971b. A comparison of two synchronizing concepts. (November). In *Acta Informatica 1*, 3 (1972), 190–199.

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

Naur, P. 1966. Program translation viewed as a general data processing problem. *Communications of the ACM 9*, 3 (March), 176–179.

Wirth, N. 1971. The programming language Pascal. *Acta Informatica 1*, 1, 35–63.