Operating systems principles and undergraduate computer science curricula*

by PETER J. DENNING

Princeton, University
Princeton, New Jersey

INTRODUCTION

In the years since 1969, the study of computer systems has assumed a role nearly equal in importance to "theory of computation" and "programming" in computer science curricula. In contrast, computer systems was regarded as recently as 1965 as being inferior in importance to these two more traditional areas of study. This is a significant change in attitude. The harbingers of the change appear in ACM's Curriculum 68,[1] and the speed of its development is demonstrated in the report of Task Force VIII of the COSINE (Computer Science in Electrical Engineering) committee of the Commission on Education of the National Academy of Engineering, entitled "An Undergraduate Course on Operating Systems Principles."[2]

The reasons for this change, the nature of computer-system studies, and the potential impact on future developments will be analyzed in this paper. I shall begin with a brief overview of computer science itself, in order to place the study of operating systems in perspective; I shall then discuss the general need in the computer industry for principles, many of which relate to computer and operating systems; finally, I shall discuss how an operating systems principles course may be organized.

ON COMPUTER SCIENCE

Computer science is the study of the design, analysis, implementation, and application of information-processing procedures.[1,3,4,5,6] In their writings, our educational leaders have consistently stressed the importance of *concepts* and *first principles* in studying the material of computer science. All proposals for computer science curricula—especially undergraduate curricula—have distinguished carefully between courses dealing with first principles and courses dealing with applications. Courses of the former type tend to be considered as "core curriculum" courses, but not courses of the latter type. (It is interesting to note that courses on operating systems were, until very recently, considered as being of the latter type. Even as it has been discovered that operating systems has a set of first principles independent of details depending on the technology, this subject matter has increasingly been considered as worthy core material.)

Courses offered in computer science curricula can be classified (roughly) into four categories: formal systems, programming systems, computer systems, and applications. The distribution of courses among the four areas will depend on the objectives of a given institution. (I believe they should be of equal importance.) *Formal systems* deals with the various mathematical systems for studying computation itself. Some of the topics taught in courses of this category include: the theory of automata, the theory of formal languages, the theory of computability, the theory of complexity of computations, and the theory of numerical computations and error propagation. *Programming systems* deals with the algorithms that perform computations on a practical level. The topics taught in courses of this category include both the concepts of programming languages and the problems of implementing algorithms. Programming language concepts encompass topics like programming language control structures (e.g., sequencing, iteration, conditional transfers, assignments), representing data, use of recursion, use of subroutines, parsing, compilation, and assembly techniques. The study of algorithms includes topics like language-independent algorithm specification, data representation, analysis of algorithms and data structures, proving *a priori* the correctness of algorithms, algorithms for searching and sorting, algorithms for

* Work reported herein was supported in part by NSF Grant GY-6586.

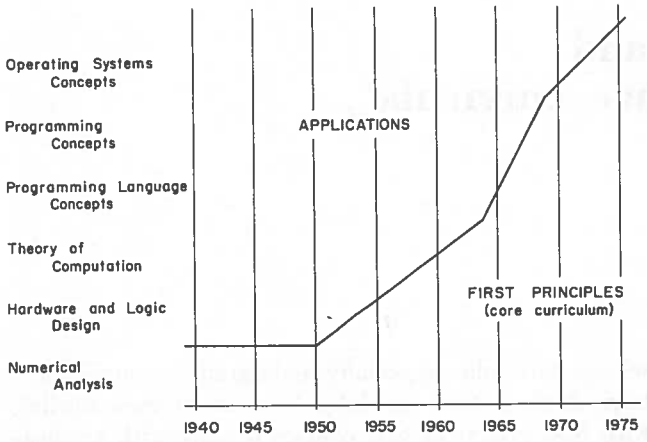

Figure 1—Evolution of first principles

dynamic storage allocation, and heuristic algorithms. *Computer systems* deals with the problems arising when algorithms are implemented on physical devices. Some of the topics taught in courses of this category include design of hardware and logic circuits, organization and design of standard equipment (e.g., processors, memories, peripherals), design of software systems, design of complex systems, control of parallelism and concurrency, control of processor and memory and other resources, analysis of system behavior, modeling the behavior of computing processes. As will be discussed shortly, these three categories deal with what I call the "first principles" of computer science. Topics which are dependent on the present-day technology, or are likely to be of little interest in more than, say, five years, are the subject of courses in the *applications* category; these include discussions of particular languages or particular machines or particular operating systems, systems programming techniques, specialized programming techniques, business data processing, and information retrieval.

The reader will note that Artificial Intelligence has not been mentioned in the above. In many respects, it cuts across all the categories. In many respects, it is an entirely different approach to computer science than the one I have outlined. Many computer science departments offer Artificial Intelligence courses on an elective basis and do not consider them as part of the core curriculum.

When I use the term "operating systems principles," I mean the software, control, modeling, and analysis aspects of the computer-systems part of computer science.

Figure 1 is a very idealized overview of the evolution of the first principles of computer science. Topics are listed along the vertical axis, time along the horizontal; those topics below the curve at a given time are considered as core-curriculum topics at that time. Conceptual approaches to numerical analysis have been with us since the beginning of electronic computing, since these were the primary purposes to which Burks, Eckert, Goldstine, Mauchly, von Neumann and their contemporaries envisioned their machines being applied. Although hardware and combinational logic design concepts using Boolean algebra to describe switching functions had been proposed by Shannon before 1945, this area received a strong impetus from the sequential machine work of Huffman, Mealy, and Moore during the early 1950's. The modern approach to the theory of computation, which emphasizes the relations among abstract machines and abstract languages and computational complexity did not evolve until the mid 1960's and the first texts in this area did not appear until the later 1960's.[7,8] Programming languages were introduced in the later 1950's (FORTRAN in 1956, ALGOL in 1958) but the concepts underlying them were not put down in systematic, textbook form until the mid-1960's.[9] Concepts about algorithms themselves, concepts more or less independent of programming languages, did not become systematized until the late 1960's.[10] And finally, the concepts of computer systems are only recently being put in text form at a graduate level[11] and nothing save a report exists describing them at an undergraduate level.[2] Thus it is evident that the development of a wide range of core material—the first principles of computer science—is quite recent in the era of electronic computing. Since much of the material of major interest to the industry itself—programming and system concepts—has just recently entered the "core area", it is no surprise that the industry has long regarded computer science as being somewhat irrelevant. I do think we can expect a change in this attitude.

On the basis of Figure 1, one can see why the majority of computer science departments were not established until after 1965. Academia has always been reluctant to establish degree programs in subject areas with no discernible broad base of first principles.

ON THE NEED FOR PRINCIPLES TO BE APPLIED IN THE COMPUTER INDUSTRY

As mentioned, Figure 1 illustrates why the industry has tended to hold computer science education in disregard, i.e., because the topics of interest to it (concepts of programming and design of systems) have not been treated on a sound conceptual basis in curricula. Looking at Figure 1 from a different perspective turns up another interesting observation: For some twenty

years, the leaders of industry (management) have had to proceed without a set of first principles in the areas of major interest to them. This perhaps explains (in part) why so much difficulty has been experienced in getting "third generation" computer systems operational: There had been no systematic set of concepts according to which designers and programmers could be guided. This had left the industry in the rather uncomfortable situation of being accused in the public media of misusing, or condoning the misuse of, computers. Witness, for example, the great public outcry with respect to privacy and protection of information.

In many respects, then, the allegations that the computing industry is "adrift" and "unprincipled" have some basis in fact. The allegations that computer science education has not been of much help to the industry likewise have some basis in fact. We cannot, however, afford to wait ten or twenty years for the present crop of computer science students to move into the leadership positions of industry. Those presently in the leadership positions must not only be made aware of the existence of principles, but they must be made aware of the principles themselves.

In a recent statement,[12] ACM President Walter Carlson recognized this problem, saying ". . . There is too little systematic codification of fundamentals (i.e., first principles). There is almost no communication between the research people and system designers or operators. There is a widening sense of frustration among academics over misuse (or lack of use) of proven computer technology in practical operations." Figure 1 demonstrates that at least a basis exists for reversing the trend addressed by Carlson's first point.

To illustrate Carlson's third point, let me discuss three examples of common misconceptions attributable to a lack of understanding of the first principles of computer science: (1) The "software problem," i.e., the increasingly high costs of software development and lack of quality control over software packages, is a long way from being solved. (2) The "protection problem," i.e., that of guaranteeing the privacy, security, and integrity of information, is yet to be solved adequately. (3) The "computer utility" was a fad which, thankfully, is dead. It would astonish some managers to know the facts: Solutions to the software and protection problem exist now—and have existed, in computer utility research projects, since at least 1965! (I will elaborate below.) While much work remains to be done, I think it is safe to say that these solutions are adequate for present-day purposes. The solutions to these problems evolved in the design and use of such systems as the CTSS (Compatible Time Sharing System) and its successor MULTICS (Multiplexed Information and Computing Service) at MIT. Both these systems are regarded as first steps in the evolution of the computer utility, and rely on the computer-utility environment for their viability; thus, the attitude that the computer utility was a fad is in fact an attitude hostile to the solutions of the two problems. That so many who hold leadership positions in the computer industry have failed to recognize that CTSS and MULTICS implement solutions to these problems—even as undergraduate students exposed to an operating systems principles course seem to have no such difficulty—further reinforces my conviction that the first principles of computer science are not widely disseminated or appreciated.

The solution to the software problem in CTSS and MULTICS is based on the concepts of *programming modularity* and *sharing of information*; the requisite mechanism took the forms of the CTSS file system and the MULTICS virtual memory.* The solution to the protection problem in MULTICS is based on the concepts *protection rings* (or "domains") and on *controlled access to information.*** An important aspect of these two solutions is, neither can be accommodated on most systems presently on the market without major alterations in hardware and software architecture. It is the failure of management to understand the principles on which the solutions are based that has made them unable to recognize that CTSS and MULTICS solve these problems, and has led them to market systems in which technically sound solutions to the software and protection problems are not possible.

The foregoing criticisms of management are meant to be constructive in the sense that managers (and other elements of the computing profession as well) cannot play their roles to perfection without a thorough understanding of the principles of computer science. And, because it is only recently that the first principles have become systematized, it is difficult to expect managers to interpret new developments in the light of guiding principles.

ON THE IMPORTANCE OF COMPUTER SYSTEMS CONCEPTS

I have devoted a considerable amount of space to emphasizing the first principles of computer science and

* A description of the CTSS file system can be found in Wilkes' remarkably concise 96-page book,[13] a description of the MULTICS virtual memory can be found in the paper by Bensoussan, Clingen, and Daley.[14] Both these descriptions make it adequately clear that these systems, in their own contexts, have solved the software problem.

** The concepts of protection systems are discussed by Lampson,[15] and by Graham and Denning;[16] systems implications of protection are discussed by Wilkes[13] and by Schroeder and Saltzer.[17]

analyzing the effects in the computer industry of having no first principles. I have done this to emphasize how the first principles of computer systems have a natural place in computer science. Aside from these larger considerations, there are several reasons why a systematic presentation of the first principles of computer systems is important in and of itself.

First, the so-called case-study approach to teaching computer systems has been more or less unsuccessful wherever it has been tried: The student becomes so immersed in the particulars of the given system that he cannot distinguish important principle from irrelevant detail. In my limited experience, I have met only with success using the "modeling and analysis" approach. Thus, computer system principles are useful in the practical task of teaching others about computer systems.

Second, it is increasingly evident throughout the computer industry that we literally can no longer afford to develop ever larger, ever more complex systems without solid notions of how they are to be structured or how they are to behave. We can no longer afford to put together systems which are extremely wasteful of their own resources. We can no longer afford *not* to apply the concepts and principles in practice.

Third, for some inexplicable reason, the design of complex software and hardware systems has been considered traditionally as an "application" problem, a "technology-dependent" problem. In contrast, it is now being realized that designing complex systems—systems which operate as intended, whose correctness and behavior can be established *a priori*, whose performance can be monitored easily—is an intellectual task worthy of our best minds. (In other words, the computer-systems area has inherited one of the most important problem areas of computer science.) This realization has stimulated increasing amounts of research in computer system modeling and analysis, a principal result of which has been the emergence of a teachable, comprehendable body of operating systems principles. Much of "computer system theory" is concerned in one way or another both with managing complexity in software and hardware systems and with complex algorithms involving parallel processes, so that it is relevant to many practical problems now facing the industry.

Fourth, and perhaps of the most long-term significance, there is an ever widening appreciation of the view that our world has become a vast real-time system whose complexity is beyond the reach of the unaided human mind to understand. It is not difficult to identify the essential elements of information systems in business systems, economic systems, urban systems, and even social systems. Again and again we see decisions being taken to regulate such systems as these which, despite the best of intentions, often turn out to have the opposite of the intended effect. Jay Forrester has called this phenomenon the "counterintuitive behavior" of complex systems.[18] It is explained by the inability of the unaided mind fully to grasp why the long-term effect of a policy is often the opposite of its short-term effect, or fully to comprehend the complexities of, and interrelations among, the various parameters that influence a system. Forrester's simulation experiments of urban and business systems show that the intended result may normally be obtained only when *all* the controllable parameters of a system are governed by a single policy, not when each such parameter is controlled individually. This type of phenomenon is hardly new in the experience of computer system designers. As computer system "theorists" axiomatize information systems, developing systematic approaches both for managing complexity and for guaranteeing *a priori* that a system will operate as intended, the results of their efforts should be applicable to the solutions of problems in social, urban and other noncomputer information systems.

ON THE APPROACH

As I have stated, I call this the modeling-and-analysis approach to studying operating systems, to distinguish it from the case-study approach. The material can be organized and presented as a sequence of abstractions together with examples of their applications, each abstraction being a principle from which most implementations can be deduced. The concept-areas (in which teachable abstractions exist) are:

procedure implementation
concurrent processes
memory management
name management
resource allocation
protection

Experience shows that the most coherent treatment is obtained if the topics are organized according to this list of concept-areas.

The COSINE report[2] goes into some detail instructing instructors how to present these abstractions; a COMPUTING SURVEYS paper[19] explores them in some detail from a student's viewpoint, the Coffman-Denning text[11] treats the mathematical analysis related to them. Accordingly, I shall restrict myself here to outlining their content.

The background required of students undertaking

this type of course should include: a working knowledge of programming language features; an elementary understanding of compilation and loading processes; processor organization and operation, including interrupts; memory organization and operation; and data structures such as stacks, queues, arrays, and hash tables. This background could be obtained from a course on programming languages (e.g., ACM's Course I-2)[1] and a course on computer organization (e.g., ACM's Course I-3).[1] A data structures course (e.g., ACM's Course I-1)[1] is helpful but not necessary.

The course itself is related to ACM's systems programming course (ACM Course I-4)[1] but differs in at least two significant ways. First, the ACM outline suggests a "descriptive," case-study approach whereas this course is organized along conceptual lines. Second, ACM's course emphasizes techniques of systems programming whereas this course emphasizes the principles of system organization and operation. This shift in emphasis is made possible by new developments, since the ACM report predates the appearance of much of the modeling and analysis material on which this course is based.

The instructor of this course will find it useful to introduce the subject by pointing out how, despite the wide range of operating systems types and capabilities, there is a set of characteristics common to these systems. These characteristics include: (1) *concurrency*, i.e., many activities proceeding simultaneously, (2) *sharing of resources* and the existence of a centralized resource allocation mechanism, (3) *sharing of information*, (4) *protection* for information, (5) *long-term storage* of information, (6) *multiplexing*, i.e., the technique of switching a resource (rapidly) among many requestors so that it is assigned to at most one at a time, (7) *remote conversational access*, (8) *nondeterminacy*, i.e., unpredictability of the order in which events will occur, and (9) *modularity* in design and operation of systems. It is important to point out that, for pedagogic reasons, the course material is presented (more compactly) according to the six concept-areas stated earlier, and not directly among the lines of these nine common characteristics.

The area of procedure implementation is important because the reason computer systems exist at all is to provide an efficient environment for executing programs. The notion of procedure is important because of its close relation to programming modularity. The basic concepts—pure procedure, procedure activations and activation records, parameters, local and nonlocal references—should be presented first. Then the main concept, "procedure in execution" (i.e., a procedure which has been called but has not yet returned) and its implementations, is presented. An abstract description of "procedure in execution" is a pair of pointers (i, r) where i is an instruction pointer and r an activation-record pointer (local environment pointer). Every implementation must solve three problems: (1) allocating and freeing storage on procedure call and return, (2) interpreting local references, i.e., those to objects in the activation record, and (3) interpreting nonlocal references, i.e., those to objects in other activation records. The implementations of "procedure in execution" in languages like FORTRAN, ALGOL, or PL/I can be deduced from these concepts by considering the restrictions imposed by these languages.

The area of concurrent (parallel) processes is important because one of the purposes of an operating system is controlling many, independently-timed activities. A "process" can be regarded as a "program in execution" and has an abstract description much like "procedure in execution." "Parallel Processes" is the notion that, at any given time, more than one program will be observed to be somewhere between its initiation and termination points. There are four process control problems of principal concern: (1) *determinacy*, i.e., the property that the result of a computation by cooperating processes on common memory cells is independent of their relative speeds, (2) *freedom from deadlock*, i.e., the property that allocation of resources is controlled so that at no time does there exist a set of processes, each holding some resources and requesting additional resources, such that no process's request can be satisfied from the available resources, (3) *mutual exclusion*, i.e., the property that, of a given set of processes, at most one is executing a given set of instructions at any time, and (4) *synchronization*, i.e., the property that a process, whose progress past a given point depends on receiving a signal from another, is stopped at that point until the signal arrives.

The area of memory management is important because every practical computer system incorporates a hierarchy of storage media characterized by various compromises among access time, capacity, and cost; a set of policies and mechanisms is required to increase system efficiency by arranging that the most frequently accessed information resides in fast access memory. The abstractions used here are those of "virtual memory": address space, memory space, and address map. The common implementations of virtual memory (e.g., paging, segmentation) can be deduced from these abstractions by considering factors such as efficiency of mapping operations and efficiency of storage utilization. Once the implementations have been studied, one can study the policies for managing them. Finally, it is straightforward to generalize the discussion to the implementations and policies of multiprogramming and of auxiliary memory problems. The foregoing develop-

ment will lead to a computational storage system (i.e., that part of the memory system in which references are interpreted by the hardware) which presents to each programmer a large, linear address space.

The area of name management is important because a linear address space such as provided by the preceding development has inherent limitations from the standpoint of programmers and system users. It cannot handle growing or shrinking objects, provide different contexts in which processes may operate, allow for sharing or protecting objects, or implement a long-term storage system in which objects may reside independently of any context. In other words, linear address space cannot support modular programming to the extent required by today's system objectives. These limitations can be overcome by extending the memory system to allow programmers to define objects of variable size, assign names and (variable) contexts to these objects, allow shared access to objects, and specify dynamically which subset of a universe of objects should participate in a computation. That part of the memory system which implements these new objectives is called the "long-term storage system"; it may be distinct from the computational storage system or it may be one and the same. Computers implementing a virtual memory and a file system are examples of the former, computers implementing segmentation are examples of the latter. In either case, a global (system-wide) scheme for naming objects must be devised (in order to allow sharing), the (directory) tree hierarchy with "pathnames" being most common (in this case the long-term storage system provides a tree-structured space of objects in addition to the linear space or spaces provided by the computational storage system).

The area of resource allocation is important partly because the complexities of the interactions among all the processes in the system dictate that a central policy regulate resource usage to optimize performance, and partly because one effect of implementing the previous abstractions is to hide machine details from users, placing the burden of allocation decisions squarely on the system. One can distinguish long-term from short-term policies, the latter being of primary concern here. One can model a process (from a resource-allocation view) as cycling among the "demand-states" ready, running, blocked; correspondingly one finds a network of queues with feedback in the computer system. Processes are distributed among the queues according to demand-states, and change queues according to changes in demand-state. In terms of this, one can study specific queueing policies and overall network control policies; one can study the meaning of concepts like "system balance" and "thrashing"; and one can study what evaluation techniques are applicable. Models of program behavior and the use of statistical analysis are important to an understanding of resource allocation and are used throughout.

The area of protection is important in any system where there may reside procedure and data belonging to more than one individual. It must not be possible for one process to disrupt or corrupt service to others, and access to information (especially if confidential or proprietary) must be permitted only under appropriate authorization. The abstract model of a protection system includes: (1) a set of "domains" or "protection contexts," i.e., sets of constraints according to which a process may access objects, (2) a set of "objects," i.e., everything to which access must be protected or controlled (including the domains), and (3) a mechanism that both specifies and enforces access rules by processes to objects, the type of access depending on the domain with which the process is associated. The mechanism can be specified in the form of an "access matrix" (e.g., if A is the access matrix, $A[d, x]$ specifies the types of access a process associated with domain d has to object x). Most of the implementations of protection found in practice can be deduced from this model.

There remain certain issues that have not been treated definitively in the literature but which nonetheless are of central importance in computer system operation and design. These include: reliability, performance evaluation, design methodologies, and implementation strategies. They must, unfortunately, be relegated to a pedagogically inferior position at the end of the course. I should emphasize that this reflects the absence of teachable material, not the importance of the issues. As viable abstractions in these areas are evolved, they will be welcome additions to the course.

CONCLUSIONS

Operating systems principles can be regarded as the study of complex algorithms comprising parallel activities. This paper has reviewed and analyzed the importance of a significant change in attitude: The assumption of computer operating systems principles into the core of computer science. The analysis of this change was done in the light of the evolution of the "first principles" of computer science, of the need for these principles to be applied in the computer industry, and of the increasing need for systematic ways of dealing with complex, real-time information systems. An outline for a course on operating systems principles was described, the concepts being chosen for inclusion in the course on the basis both of demonstrated utility in practice and of their being straightforward generalizations of widely accepted viewpoints.

The first twenty-five years of the computer industry, years of remarkable achievement, have not been without their problems. Now that computer science education is maturing, we should be able to expect closer cooperation between universities and industry in solving these problems.

REFERENCES

1 ACM Curriculum Committee on Computer Science (C³S)
Curriculum 68: Recommendations for academic programs in computer science
Comm ACM 11 3 March 1968 151-197

2 COSINE Task Force VIII
An undergraduate course on operating systems principles
(The members of the task force were: Peter J. Denning, Jack B. Dennis, A. Nico Habermann, Butler W. Lampson, Richard R. Muntz, and Dennis Tsichritzis.) June 1971. Available free of charge from: Commission on Education National Academy of Engineering 2101 Constitution Avenue NW Washington DC 20418

3 S AMAREL
Computer science: a conceptual framework for curriculum planning
Comm ACM 14 6 June 1971 391-401

4 G E FORSYTHE
A university's educational program in computer science
Comm ACM 10 1 Jan 1967 3-11

5 R W HAMMING
One man's view of computer science
J ACM 10 1 Jan 1969 3-12

6 A L PERLIS
University education in computing science
Proc Stony Brook Conf Academic Press 1968 70ff

7 J E HOPCROFT J D ULLMAN
Formal languages and their relation to automata
Addison-Wesley 1969

8 M M MINSKY
Computation: Finite and infinite machines
Prentice-Hall 1967

9 I FLORES
Computer programming
Prentice-Hall 1966

10 D E KNUTH
The art of computer programming
Addison-Wesley Vol I 1968 Vol II 1969

11 E G COFFMAN JR P J DENNING
Operating systems theory
Prentice-Hall to appear

12 W M CARLSON
President's letter: reflections on Ljubljana
Comm ACM 14 10 Oct 1971

13 M V WILKES
Time sharing computer systems
American Elsevier 1968

14 A BENSOUSSAN C T CLINGEN R C DALEY
The MULTICS virtual memory
Proc 2nd ACM Symposium on Operating Systems Principles Oct 1969 30-42

15 B W LAMPSON
Protection
Proc 5th Annual Princeton Conference on Information Science and Systems Department of Electrical Engineering Princeton University Princeton New Jersey 08540 March 1971

16 G S GRAHAM P J DENNING
Protection: principles and practice
Proc AFIPS Conf 40 Spring Joint Computer Conference 1972

17 M D SCHROEDER J H SALTZER
A hardware architecture for implementing protection rings
Comm ACM 15 3 March 1972

18 J W FORRESTER
Urban dynamics
MIT Press 1969

19 P J DENNING
Third generation computer systems
Computing Surveys 3 4 Dec 1971