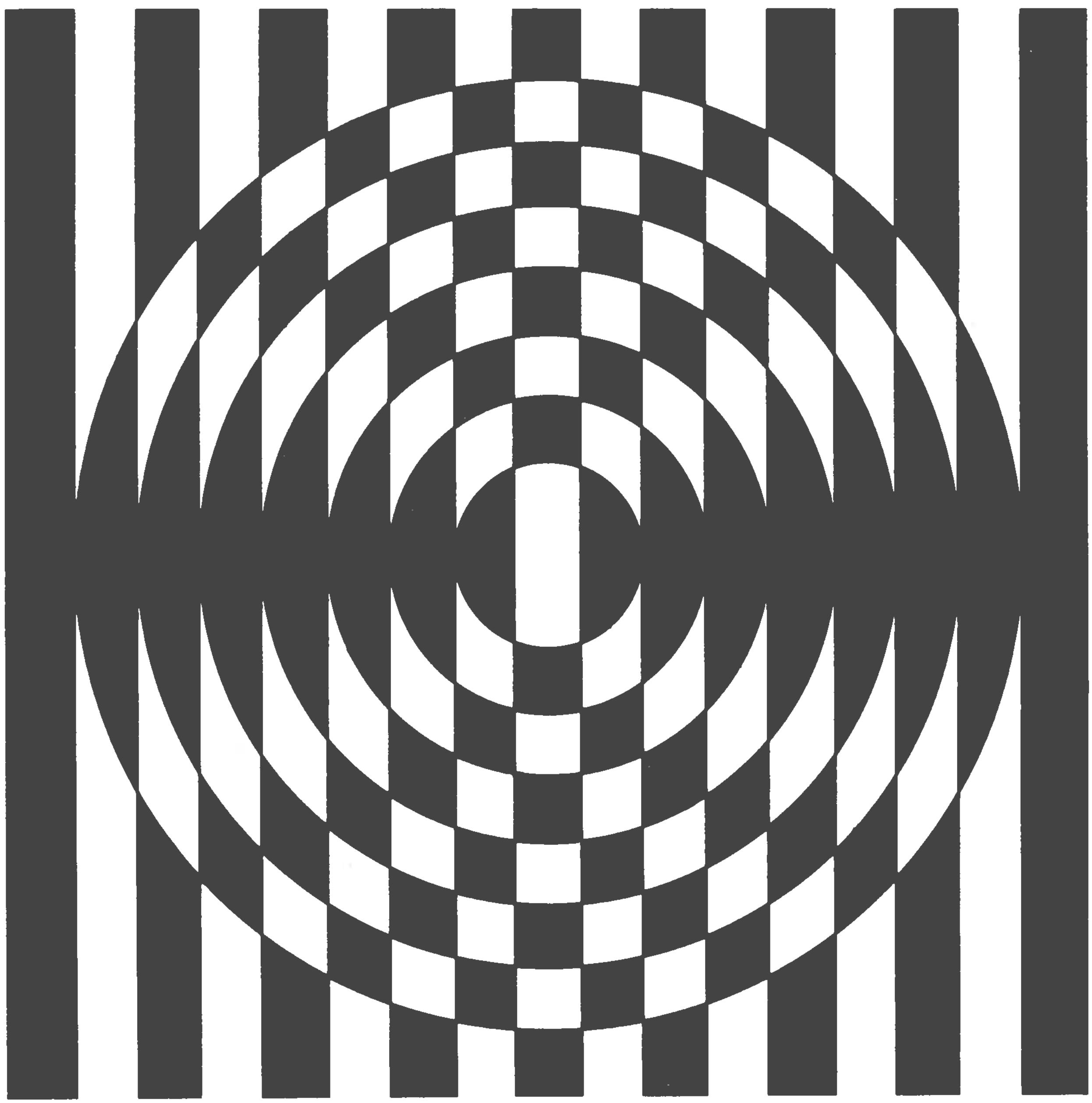
An Undergraduate Course on Operating Systems Principles

June 1971

Cosine Committee

Commission on Education



COSINE TASK FORCE PUBLICATIONS:

Task Force I	Some Specifications for a Computer-Oriented First Course in Electrical Engineering. September, 1968.
Task Force II	An Undergraduate Electrical Engineering Course on Computer Organization. October, 1968.
Task Force III	Some Specifications for an Undergraduate Course in Digital Subsystems. November, 1968.
Task Force IV	An Undergraduate Computer Engineering Option for Electrical Engineering. January, 1970.
Task Force V	Impact of Computers on Electrical Engineering Education—A View from Industry. September, 1969.
Task Force VI	Digital Systems Laboratory Courses and Laboratory Development. March, 1971
Task Force VII	In Preparation.
Task Force VIII	An Undergraduate Course on Operating Systems Principles. June, 1971.

Available at no charge on request from:

Commission on Education
National Academy of Engineering
2101 Constitution Avenue, N.W.
Washington, D.C. 20418
Tel: (202) 961-1417

These reports have been prepared under the auspices of the Commission on Education of the National Academy of Engineering. Commission policy is to encourage the exploration of new ideas in engineering education. The Commission has been kept informed of the discussions of the COSINE Committee but has taken no position on its reports or recommendations.

The work of the COSINE Committee and these publications are supported in part by the National Science Foundation under Contract NSF-C310, Task Order No. 161.

ON OPERATING SYSTEMS
AN UNDERGRADUATE COURSE PRINCIPLES

PROLOGUE

Computer-based information systems increasingly influence our lives. Computers are no longer regarded simply as ultra-fast calculating assistants for engineers and scientists, or as fanatically accurate clerks for business. Computer systems are becoming storehouses for an enormous variety of information both public and private, and repositories for a vast resource of algorithms devised painstakingly by the practitioners of all disciplines. The presence of computer information systems is evident to anyone with money in a bank, who has been a passenger on an airline flight, or who is even casually acquainted with the space exploration program or military operations.

Even as the demand for people with a strong conceptual understanding of issues arising in computer-based information systems continues to expand, the educational programs existing in most of our universities are woefully inadequate with respect to providing professional workers in this important field of computer systems design and application.

The subject of computer operating systems, if taught at all, is typically a descriptive study of some specific operating system, with little attention being given to emphasizing the relevant basic concepts and principles. To worsen matters, it has been difficult for most university departments to develop a new course stressing operating systems principles partly because the best people in the field are often attracted to lucrative commercial opportunities, and partly because there are essentially no suitable textbooks on the subject. The best source material is found in technical papers that frequently are hard to locate, understand, or correlate.

In view of these circumstances, Task Force VIII of the COSINE Committee was established to formulate subject matter for an undergraduate elective subject on computer operating systems principles for students whose major interest is in the engineering of computer systems and software. The members of Task Force VIII were:

Peter J. Denning, Chairman	Princeton University
Jack B. Dennis	Massachusetts Institute of Technology
Butler Lampson	Xerox Research Laboratory, Palo Alto
A. Nico Haberman	Carnegie-Mellon University
Richard R. Muntz	University of California, Los Angeles
Dennis Tsichritzis	University of Toronto

Plan of the Report

The structure of this report has been motivated by two considerations. First, the students taking the course should be provided with an intellectual basis adequate for understanding and designing operating systems five and ten years in the future. Second, the universities using this report will vary considerably with respect to course duration, pace at which new concepts can be introduced, and student preparation. For these reasons we have organized the material into eight "modules," each dealing with material forming an important conceptual component of current knowledge. Each module contains these parts: a model for some aspect of operating systems, a description of the features of the model and its manifestations in specific systems, a topic outline, and a guide to the literature. We have tried to give more complete discussions of those parts of the course where, in our opinion, the available literature is most inadequate.

We expect that the instructor using this report will select topics from the modules according to his students' needs, their level of background and experience, and their ability to absorb the more advanced material.

The modules are:

1. Introduction
2. Procedures
3. Processes
4. Memory Management
5. Name Management
6. Protection
7. Resource Allocation
8. Pragmatic Aspects

Computer systems take many forms according to their function, and are controlled by a corresponding variety of operating systems ("contol," "supervisor," or "executive" programs). All these systems have certain common characteristics, and certain common major issues that must be resolved if system designers are to achieve a practical result. These common properties are the subject of Module 1.

An operating system together with the processing and memory hardware on which it runs, constitutes an environment for running users' programs, as well as an environment within which data bases and libraries may reside. The most fundamental aspect of a computer system is the application of an algorithm or procedure to data to produce a desired effect. It is important that the student understand the concentual basis for the common methods of implementing procedure application. This is the subject of Module 2.

By their very nature, operating systems involve concurrent activities. For efficient resource utilization, input and output activities are performed simultaneously with program execution. Multiprogramming is used to achieve better use of processor and memory capacity by switching between programs whenever the program being executed comes to a temporary pause. Since concurrent activities are represented in contemporary systems by processors executing separate sequential programs, the study of interacting sequential processes is fundamental to an understanding of operating systems. The notions of sequential processes and their interaction forms the subject matter of Module 3.

Every practical computer system incorporates several varieties of physical storage media, characterized by different compromises among access time, capacity, and cost. Memory management is concerned with increasing the system's efficiency by arranging that the most frequently accessed information resides in fast access memory. As multiprogramming has become more important, and as more importance has been attached to ease of programming, so the trend has been toward management of memory by the operating system rather than by user programs. These issues are treated in the context of current memory technology in Module 4.

Modular programming increases the ability for a user to construct larger programs from component subprograms without requiring that the user know the internal operation of the components. The extent to which a system can support modular programming will depend on its ability to deal with names (identifiers) in varying contexts and on its ability to allow shared access to information, both abilities being related intimately to the conventions used for handling names. These issues are the subject of Module 5.

The need to protect arises as soon as a computer system holds procedures and data belonging to more than one individual. It must not be possible for one user's actions to disrupt or corrupt service to other users. Access to procedures and data, especially if confidential or proprietary, must be permitted only with appropriate authorization. The principles underlying implementations of protection form the subject matter of Module 6.

Resource allocation is concerned with obtaining optimal utilization of system resources (processor, memory, auxiliary storage, and so on) toward meeting the system's operating objectives (throughput, response times, minimum cost, and so on). Models of program behavior and the use of statistical analysis are important to an understanding of resource allocation. This is the subject of Module 7.

There remain certain issues concerning computer system operation and design, issues that have not yet been analyzed definitively in the literature but nonetheless are very important. These include reliability, design methodologies, implementation strategies, and performance evaluation. They are the subject of Module 8.

Relation to Previous COSINE Work and ACM Curriculum 68

The course proposed in this report is designed as an advanced course to follow basic courses on computer organization and programming languages. The COSINE Committee report of September, 1967 recommended that the operating systems course be regarded as elective. In contrast, Task Force VIII recommends that this course be considered, as much as possible, an integral part of a computer science program (whether it is a core course will depend on the needs and resources of a given department). We are able to recommend such an increase in the importance of this course because there now exists a much sounder conceptual basis for teaching the principles of operating systems than existed as recently as 1969.

With respect to the ACM computer science curriculum 68 (Comm. ACM 11, 3, March 1968), it was not the intention of COSINE to "implement" any course in ACM's curriculum. Although the COSINE course is related to ACM's course I4 (systems programming), it differs in at least three significant ways. First, the ACM course description is an outline, whereas the COSINE course is a detailed specification. Second, the ACM outline suggests a descriptive, "case-study" approach, whereas ours is organized along conceptual lines. Third, ACM emphasizes the techniques of systems programming, whereas COSINE's emphasis is on the principles of system organization and operation. This shift in emphasis has been possible because the members of the task force have been associated closely with current work on advanced operating systems, the conceptualization of which has taken place since the preparation of the ACM report.

The material outlined in the Background section of this report is organized along the lines of courses I2 (programming languages) and I3 (computer organization) of the ACM curriculum. Familiarity with a course such as I1 (data structures) is desirable though not necessary. It must be emphasized that these ACM courses cover mcuh more than is required as background for this course. It must be emphasized also that these particular ACM courses are cited as examples of possible background courses; the implementation of this course does not depend on prior implementation of the ACM courses.

Project

The committee recommends strongly that the conceptual and theoretical material outlined in this report be accompanied by a reasonably detailed study of some particular operating system embodying these concepts. Although the abstractions used in the various modules of this report serve to provide the student with an understanding of the principal components of an operating system, they will do little to instill insight into how the different mechanisms mesh into a working whole or into how complexity is engendered. The instructor may wish to draw examples from a number of different systems, but the committee believes that the students should be given the opportunity to understand one complete design.

The system to be studied in depth should not be too large and it should have some coherence of design, so that the student will not be overburdened with irrelevant detail. On the other hand, the system should have sufficient scope to illustrate the essential ideas of the course: a monoprogramming batch system would, for example, be unsatisfactory. Finally, the system should be documented adequately, so that recourse to the operating system code is not necessary for a detailed understanding of its implementation. The committee is aware of only a few systems that meet these requirements. These are listed below, together with citations of their documentation and addresses from which the documentation can be obtained. Since the supplies of the documentation are limited, the committee suggests that the instructor obtain one copy and arrange with the source of the document for permission to reproduce enough copies for his class.

1. RC-4000 Software: Multiprogramming System. (P. B. Hansen, Ed.) A/S Regnecentralen
   Copenhagen, Denmark

2. Cal 6400 Time Sharing System
   Director, Computer Center
   University of California
   Berkeley, California 94720

3. CLICS: Classroom Information and Computing System (a simplified version of Multics), Rpt. MAC-TR-80.
   Project MAC Document Room
   545 Technology Square
   Cambridge, Mass. 02139

Using the References

In order to avoid overburdening a prospective instructor or his students, we have paid a great deal of attention to limiting the size of our bibliography. A citation has been included only if it satisfies, in our opinion, one or more of three criteria: 1) it is most relevant to the discussion in the modules, 2) it is the only citation available, or 3) it contains clear exposition and a good bibliography of its own. The bibliography appears at the end of the report. Each citation is of the form [*i*, name] where *i* is the index key in the bibliography and "name" designates the author or authors. At the end of each module we have included a *Reference List Guide* which summarizes the citations in that module together with indicators of three kinds:

type: C-conceptual, D-descriptive, E-example, T-tutorial
level: S-student, A-advanced student, I-instructor
importance: an integer 1-5 indicating the relative importance of the reference to the module; integer 1 is most important, 5 least.

Thus, if [*i*, name] is flagged with (CD,A,3), it is both conceptual and descriptive, appropriate reading for advanced students, and of importance rank 3. A given citation may appear in several modules with different indicators, as appropriate for that module.

MODULE 0 — BACKGROUND

The student of computer operating systems should have a good understanding of 1) programming languages, 2) computer processor organization, 3) memory organization and 4) data structures. The discussion to follow indicates the required level of maturity in these areas. Though not all the material is essential, if any significant part is unfamiliar to the student he is not prepared for the course. Deficiencies in background can be remedied through a self-study review at the beginning of the course.

0.1 Programming Languages

It is essential that the student be experienced in symbolic programming and be familiar with important language features such as expressions, data types, data structures, procedure application, formal and actual parameters, and recursion. He should understand the notion of program modularity, the idea of subprograms which can be used without knowledge of their internal structure or operation. He should also understand how source language statements and machine code are related; in particular he should understand the functions of the assembly, loading, and execution of a program, from the user's standpoint. Discussions of these topics can be found in [40,Hellerman, Ch. 2] and [33, Gear, Chs. 3-4].

0.2 Processor Organization

The student should be familiar with at least one computer processor and its instruction set. His understanding should embrace hardware functions more deeply than implementation details; it should encompass how processor features, such as index registers and indirect addressing, relate to the implementation of programming language features. See [40, Hellerman, Secs. 8.1-8.6], [33, Gear, Ch. 2] and [5, Bell].

It is particularly important that the student have a basic knowledge of interrupt mechanisms. [40, Hellerman, Sec. 8.12], [33, Gear, Sec. 2.5], and [5, Bell]. This should include the types of interrupts commonly provided, the control (enable, disable) of interrupts, and the distinction between the arming and firing of an interrupt. It includes comprehension of the distinction between interrupts originating in other processors (e.g., I/O interrupts) and traps or faults originating in the processor itself (e.g., invalid instructions).

0.3 Memory Organization

The student should be familiar with the common memory types (integrated circuit, core, mass or bulk core, disk, etc.) and appreciate relative costs, capacities, and access times [40, Hellerman, Secs. 3.1-3.2], [33, Gear, Secs. 6.1-6.4], [5, Bell], and [75, Sharpe]. He should understand the distinction between sequential access and random access devices, especially in terms of latency time characteristics. He should understand the properties of associative memory.

The student should understand that a channel is a special-purpose processor, and how communications between central processors and channels are effected. [40, Hellerman, Sec. 8.13], [33, Gear, Sec. 6.5], and [5, Bell].

0.4 Data Structures

The student must understand the most common types of data structures and their representations using both sequential and linked allocation; these include stacks, queues and arrays [50, Knuth]. He should be familiar also with hash tables [63, Morris].

Module O: Background — Topic Outline

0.1 Programming Languages

- Working knowledge of symbolic progamming features
  - Data types
  - Variables
  - Expressions
  - Procedure application
    - Formal parameters
    - Actual parameters
    - Recursive procudures
- Machine language representation of source language statements
- Simplified program "history"
  - Assembly or compilation
  - Loading
  - Execution

0.2 Processor Organization

- Machine language concepts
- Relationships of processor features to programming language features
- Implicit control
- Data movement
- Data transformation
- Program control
- Addressing
  - index registers
  - indirect addressing

0.3 Memory Organization

- Hierarchy of memory types—core, drum, disk, tape and other mass media
    - cost
    - capacity
    - access times
- Concepts of random access, direct access and sequential access devices
- Associative memories
- I/0 control, channels, CPU communication

0.4 Data Structures

- Stacks, queues, arrays
  - Sequential allocation
  - Linked allocation
- Hash tables

Module O: Background — Reference List Guide

types: C — conceptual, D — descriptive, E — example, T — tutorial

level: S — student, A — advanced student, I — instructor

Key	Author	Type	Level	Importance
5	Bell and Newell	T	A	1
33	Gear	T	S	1
40	Hellerman	T	S	2
50	Knuth	T	S	1
63	Morris	T	S	1
75	Sharpe	T	A	2

MODULE 1 — INTRODUCTION

1.1 Forms of Systems

Already, most students will have heard of systems which use operating system techniques extensively. These include: 1) real-time control systems: reservations, telephone switching, process control; 2) data base systems: management information, credit reporting; 3) general purpose programming systems: batch, multi-programming, time-sharing; and 4) computer networks.

By considering characteristics these systems have in common, and issues which arise over and over again in designing or understanding them, this module provides an overview of the course. The common characteristics and issues may be used as touchstones by which to motivate, and against which to judge, the ideas and techniques forming the body of the course.

1.2. Views of a System

Almost as varied as the types of systems are the views programmers and designers hold of them. These views include: 1) The system defines an extended language .2) The system defines an extended machine, e.g., a "virtual machine." 3) The system creates an environment for efficient program execution. 4) The system is an information management system. The instructor can find many other examples of viewpoints as he peruses the literature. Despite the wide variation in types of systems and views about them, they have an important and extensive set of common characteristics. These are discussed next.

1.3. Common Characteristics

The four types of systems listed above all employ some form of *concurrency*; for example, many reservation agents may be engaged in transactions at one time, many subsystems of a chemical plant must be controlled, printing and computing are overlapped in a simple batch-processing system, input-output for many jobs is processed concurrently in a multiprogramming system. In some cases the concurrent activities are almost independent (as in multiprogramming), in others they are related through a shared data base (reservations), in still others there are more complex interactions (input-output overlap in a single job, chemical processes). When the activities are independent, concurrency is the concern of the underlying system; as they become more closely related, explicit recognition of the concurrency must appear in their implementation. There are no clear dividing lines, and methods for handling concurrency must be available to both system and users.

Closely related to concurrency is *sharing of information*. Examples of information shared among many users include: a FORTRAN compiler, the records of a reservation or credit reporting system, a table of stock prices accessed by security analysts in a time-sharing system, or channel commands and status information shared between an input-output channel and a central processor. Sharing begets unique problems, the most important one being that concurrent attempts to access and modify data can lead to races or to use of data while it is in an inconsistent state.

Closely related to sharing is *long-term storage* of data in a computer system. In fact, all the examples of sharing cited above (except the last) are also examples of long-term storage. Three major problems must be solved in an implementation of long-term storage: 1) Maintaining records of the location of data, and communicating location-information to all users (file systems and naming). 2) Controlling access to data (privacy and protection). 3) guaranteeing survival of the data despite system failures (reliability).

A property resulting from concurrency and data storage is *nondeterminacy*. On the one hand, a batch-processing system for FORTRAN program is (or should be) determinate in the sense that it will give the same results every time it is run with the same inputs. On the other hand, a transaction with a reservation system is nondeterminate, since it may be in a race with another transaction for the last available space, and since its effect may depend on the state of the data base.

Sharing of resources, contrasted with sharing of information, is another important characteristic of most computer systems. Multiprogramming systems share memory, time sharing systems share the central processor, all systems share channels and disk storage. This kind of sharing is motivated by economics, i.e., the desire to reduce costs by sharing equipment. It raises special problems in protection and resource allocation.

Many systems employ various types of *modularity* in their design and operation. Here, modularity means the ability to construct complex systems from separately designed parts. It appears in several forms, particularly programming modularity and operating system functional modularity.

A final, very important characteristic of many real time, data base, and general purpose systems—and of all computer networks—is *remote conversational access*. Conversation requires a system to respond promptly and to switch (multiplex) its attention among users at a high rate, i.e., support a high degree of concurrency. Remote access requires it to interface with the telephone system and to handle large numbers of slow terminals.

1.4. Major Issues

The following list of words suggests some important concerns which intersect all the functional divisions of section 1.1:

generality
reliability
efficiency
complexity
compatibility

Except for generality, unfortunately, these are issues whose importance is not yet supported by any useful conceptual framework. As a result, we have relegated them to an inferior position at the end of the course; we emphasize that this relegation *reflects the absence of teachable material and not the importance of the issues*. A few general remarks about these issues are included below.

Generality escapes the observations of the preceding paragraph at least to some extent. Indeed, all the abstractions in the body of the course can be viewed as attempts to increase the generality of the basic mechanisms in operating systems and thus to give them wider applicability, both as aids to understanding and as tools for programming.

Reliability can be considered under several headings: 1) Coping with hardware unreliability, by reconfiguration and recovery from detected failures such as parity errors. 2) Making programs reliable by making their structure and interfaces very clear, or by proving their correctness. 3) Dealing with software errors by redundancy and recovery. Some examples of redundancy should be mentioned, e.g. the use of doubly linked lists or of 'headers' or 'home addresses' on disk records. The idea of a recovery procedure may also be clarified by an example, such as a disk file backup and reloading system.

Efficiency is partly a matter of implementation detail; as such, it should be brought out in the study of the example system. It is, however, far more a matter of conceptual organization and algorithms design. A significant reason for this (though by no means the only one) is the reduction in system size and overhead which results from good design.

Complexity is the enemy of reliability, and often of efficiency and generality as well. It is an inescapable aspect of, and indeed often the reason for, using computers. One interpretation of the purpose of this course is a description of tools for structuring complex processes in comprehensible ways.

A casual perusal of the trade literature will reveal the concern of both manufacturers and users with compatability.

Module 1: Introduction — Topic Outline

1.1. Forms of computer system (software/hardware composites)

- Real time control systems
- Data base systems
- General purpose programming systems
- Computer networks and utilities

1.2. Views of system

- Defining an extended language or virtual machine
- Establishing an environment for program execution
- Information management system

1.3. Common characteristics

- Concurrency
- Sharing
- Nondeterminate long term storage (data bases)
- Modularity
- Conversational remote access

1.4. Issues

- Reliability
- Generality
- Efficiency
- Complexity
- Compatibility

Module 1: Introduction — Reference List Guide

types: C — conceptual, D — descriptive, E — example,
T — tutorial
level: S — student, A — advanced student,
I — instructor

Key	Author	Type	Level	Importance
13	Corbató and Vyssotsky	D	A	2
14	Crisman	D	I	3
30	Fano and Corbató	D	S	1
69	Parkhill	D	A	2
75	Sharpe	D	I	3
85	Wilkes (Chs 1,2)	D	S	1

MODULE 2 — PROCEDURE IMPLEMENTATION

A theme which reappears throughout the course is: a primary purpose of an operating system is providing an efficient and convenient environment for executing programs. This module examines this view in some detail. The most fundamental aspects of procedure implementation are discussed here. Further aspects affecting the convenience with which program modules can be combined are treated in Module 5.

2.1 Abstract Model of a Procedure

A "procedure in execution" consists of: 1) instruction code representing an algorithm, 2) an activation record defining the local environment of the procedure, and 3) the nonlocal environment of the procedure. The total environment of a given procedure comprises the data structures and procedures that are currently accessible to the given procedure. The local environment of a given procedure comprises the local working storage for the current activation of the procedure. The "activation record," which is created as part of the procedure activation, will in general contain the local working storage, the values or addresses of actual parameters, the return address, and pointers to the remainder of the environment. It is important to note that the total environment is determined by the context in which the procedure is activated. Since a procedure may be activated at different points in a computation, the local and non-local environments will in general be different for each activation.

There are three basic problems that an implementation of procedures must solve. First there must be a mechanism for referencing the non-local environment, i.e. non-local variables and other procedures. Second, there must be a mechanism for activating a procedure, incorporating a way of naming the procedure to be activated, constructing its activation record, and transferring control. Third, there must be a mechanism for passing of parameters to the activated procedure.

The conceptual model described above should be illustrated by implementations found in practice. These can be taken from common programming languages familiar to the students. FORTRAN and ALGOL are used as examples in the next two sections.

2.2 Example Implementation — FORTRAN

The definition of the FORTRAN programming language gives rise to an especially simple run-time environment [77, Standards]. A FORTRAN program consists of a set of one or more disjoint procedures (subroutines). The non-local environment of each procedure consists of the other procedures and the variables in COMMON. Since the addresses of procedure entry points and COMMON variables are known by load time, references and linkages to these items can be resolved at this time before execution begins. Since FORTRAN prohibits recursive activations of procedures, the local storage of a procedure is permanently allocated and the same locations used for each activation.

Parameters are passed only by reference (i.e. addresses of parameter storage locations are passed). A common technique is to store the addresses of parameters in the successive storage locations immediately following the subroutine call instruction. Since the subroutine call instruction places in the subroutine's return address cell 1 plus its own address, the subroutine can locate its parameters by interpreting the contents of the return address cell as a pointer to the list of parameter addresses. The usual convention for returning control is to execute a jump to the first location following the parameter list. A discussion of the implementation of FORTRAN subroutines can be found in [33,Gear].

After having presented the call, return, and parameter-passing mechanisms of FORTRAN, the instructor should review the operation of linking loaders, which combine independently compiled subroutines into a program [59, McCarthy].

2.3 Example Implementation — ALGOL

To complete the discussion, the instructor should present an implementation of procedures which involves recursion, and the associated dynamic storage allocation of space for activation records. For this purpose it is sufficient to consider a subset of ALGOL in which the passed parameters are simple variables and the only external names which a procedure may use are those of other procedures. [67, Naur, et al.] A review of the nested declaration structure of ALGOL and the scope rules for identifiers should be given.

Since ALGOL permits the recursive use of a procedure, an implementation must permit two or more activations of the same procedure to exist simultaneously. This implies that each activation of the procedure be provided with a distinct activation record. The nesting of procedure activations in ALGOL makes the use of a stack convenient for this purpose. Each activation record will consist of the return address, actual parameter information, a pointer to the current activation record of the calling procedure, and the local storage for the activated procedure. The first three items can be set up by the calling procedure; the fourth can be done upon entry to the activated procedure so that a variable amount of storage can be allocated (e.g., local arrays can be of variable size). The actual transfer of control is simply a jump to the first location (entry point) in the fixed-program part of the activated procedure. As in FORTRAN, entry points of procedures will be known at load time. The location of the activation record of the currently executing procedure resides in a system base register or index register, and all local data references are taken relative to the contents of this register.

Parameters (which we have limited to simple variables) can be passed by name or by value. If a parameter is passed by name it is to be evaluated each time it is used in the procedure, this evaluation being performed in the environment of the calling procedure. In the restricted situation described here, the parameter is a variable in the local data area of the calling procedure; accordingly, a name parameter can be specified by its local address within the activation record of the calling procedure. If the parameter is passed by value, the value of that parameter is copied into the activation record. Returning control to the calling procedure consists of deleting the activation record from the stack and executing a jump to the return address.

At his option, the instructor may extend the discussion to include the ALGOL facility for referencing arbitrary non-local names (not just procedure names). To do so, he will have to introduce the concept of a "static chain" of activation records. The successive entries in this chain are the lexicographically enclosing blocks in the ALGOL declaration. (This is contrasted with the dynamic chain," whose successive entries are the activation records in order of activation; the chains may be different, for example, when recursive procedure calls have been made.) A static chain is unnecessary if the only non-local names are external procedure names because entry points are known prior to execution; but the addresses of data items within activation records are not known prior to execution.

Comprehensive treatments of ALGOL implementation can be found in [71, Randell] and [81, Wegner]. Procedure implementations are covered more generally in [25, Dennis].

Module 2: Procedure Implementation — Topic Outline

2.1. Basic Concepts

Pure procedure
Procedure activations, activation records
Parameters, formal and actual, value and name
Environment, local and total
Local and non-local references

2.2. FORTRAN Implementation

Memory arrangement for nonrecursive procedures
Transfer and return of control
Passing parameters by reference
External references, common locations
Design and operation of linking subprogram loader

2.3. ALGOL Implementation

Block structure, scope of names
Recursive procedures, activation record stack
Passing parameters
Static and dynamic chains

Module 2: Procedure Implementation — Reference List Guide

types: C — conceptual, D — descriptive, E — example, T — tutorial
level: S — student, A — advanced student, I — instructor

Key	Author	Type	Level	Importance
25	Dennis et al.	CT	I	1
33	Gear	T	I	1
59	McCarthy	CT	I	2
67	Naur et al.	D	I	3
71	Randell and Russell	DT	I	2
77	Standards (ASA)	D	I	3
81	Wegner	T	A	1

MODULE 3 — PROCESSES

The study of operating systems adds another dimension to the design of computer programs, in the form of "concurrent programming." It is desirable to add this dimension for at least two reasons:

1. the demand for a short response time which can be met by means of various forms of multiprogramming and multiprocessing;
2. efficient utilization of equipment, which can be realized by means of concurrent activity between the central machine and its peripheral devices.

Although present hardware technology makes concurrent activity feasible, it is the task of the operating system designer to write the programs that will effect it. This task separates quite naturally into two parts: writing the programs for each of the individual activities, and designing the interactions between them. A method for dealing with these two aspects, which has proved effective, is to build a system as a set of sequential processes that interact at well-defined events (see for instance descriptions of MULTICS [13, Corbató], THE [28, Dijkstra], RC4000 [36, Hansen]). Instead of restricting it to be associated with a hardware processor, a process should be regarded as an activity that executes one of the system functions, an activity which the designer wishes to be performed conceptually in parallel with other activities. Taking this point of view, a hardware processor is considered as a resource which is needed by a process to carry its activity forward in real time and which could be shared among processes. "Parallel activity" is then interpreted in the following sense: when a snapshot is taken of the system, several processes may be found somewhere between their starting point and their points of completion.

The foregoing outlines the motivation for studying parallel process as part of operating systems. The previous modules have provided the ingredients for starting a study of operating systems; the treatment of concurrent programming is the beginning of the study proper. The remainder of this module concentrates on the interaction of processes and the tools that enable parallel processes to share information. Appendix M3-B contains problems of the type often encountered in concurrent programming.

3.1 Parallelism in an Operating System

Almost independent processes operating in parallel exist already in the given hardware consisting of a central machine and its peripheral devises. Processes designed in software should also be considered as almost independent and nothing should be assumed about their relative speeds (see the lecture notes [27, Dijkstra]).

At his option, the instructor should devote some attention to an abstract description of a process as a set of histories, in which a history is a sequence of states $S_i = (P_i, M_i)$ comprising a processor state P_i and a memory state M_i (see the lecture notes [25, Dennis, chapter 7]). In terms of an abstract description the major issues of concurrent programming (common state variables, mutual exclusion, abstraction, nondeterministic processes, synchronization and deadlocks) can be elucidated (a detailed treatment is found in [41, Horning]).

The instructor should conclude this preliminary discussion by exhibiting some examples in which the problems mentioned show up in existing systems. If he has some experience with operating system design he certainly will know several examples of synchronization or deadlock problems.

3.2 Mutual Exclusion

Whenever two or more processes may access common information cells, some restrictions must be imposed on their access to such cells, for otherwise misrepresentation of information may result. The requirement that at most one process may be using a common cell at any given time is known as mutual exclusion. Implementing it requires "primitive operations" on data, where primitive means that an operation cannot be interrupted by any other operation on this data. An instructive start in writing concurrent programs is to take "copy a value of a variable" as primitive operation [26, Dijkstra]. The student will find out that this primitive, though rather clumsy, is sufficient to solve the mutual exclusion problem. Since this solution uses the busy form of waiting, there is a motive for looking for a better set of primitives. The primitives LOCK, UNLOCK and P and V operation (with the counting semaphore) should be discussed in this light; see the appendix in [28, Dijkstra]. Other forms of the same primitives should be identified, e.g. [73, Saltzer] and [86, Wirth]. Some attention should be given to implementing primitives, see Appendix M3-A and [86, Wirth]. Finally, it should be pointed out that application of these primitives may prove very unfair for some of the blocked processes unless certain priority rules are implemented, implicitly or explicitly; see Appendix M3-B(4).

3.3 Synchronization

This subject should be introduced in relation with the cooperation of processes sharing the system facilities. The cooperation implies that a process should not continue at certain points of its program until certain information is supplied by another process; moreover, the correct operation of the system usually requires that processes ought always to supply that information without which others cannot proceed. Although considered as asynchronous, processes should be synchronized up to such an extent that the two conditions, "the necessary information becoming available" and " the continuation of a process dependent upon that information," are ordered in time. In this sense the solution of the mutual exclusion problem is an example of synchronization, according to which one process requires outside information and other processes supply this information. If a given process discovers that the required information is not yet available when needed, that one process will block itself and it is the task of the others to wake it up when the required information becomes available. The primitives to be discussed here are the pairs of operations (BLOCK, WAKEUP) and (P, V). The occurrence of race conditions and the solution that uses

the "wake-up-waiting switch" deserve special attention (see the discussion on the wake-up-waiting switch [72, Saltzer]). This topic lends itself to further exercises in concurrent programming. With regard to avoiding a fixed selection scheme in a V-operation the concept of the private semaphore, i.e., one which can cause the stopping of only one given process, should be discussed [27, Dijkstra].

3.4 Process Communication

A discussion of the relation of a Sender and a Receiver communicating via a message buffer is a natural continuation of the previous section. Updating the status variables of the buffer requires mutually exclusive access and the states "buffer empty" and "buffer full" require synchronization of Sender and Receiver [27, Dijkstra].

An immediate extension to the above allows m Senders and n Receivers ($m \geq 1$, $n \geq 1$) to communicate via a message buffer; see [27, Dijkstra] and the Mailbox description [76, Spier]. A further extension allows Receivers to inspect the buffer for a message of highest priority instead of treating them in a strict first-in-first-out order. At this stage considerations of implementation must be taken into account. (See Appendix M3-A.) It should not be possible, for instance, that a subset of Senders monopolizes the buffer so that others will never get a chance to deposit a message. Moreover, when a message has been placed in the buffer, its Sender should be able to detect that it has been accepted properly. The RC4000 is a system having these facilities [37, Hansen]. These considerations show that the primitives are adequate tools, but not more than that, it being still the designer's task to effect the correct cooperation of processes.

3.5 Switching Control

The object of this section is discussing how a set of cooperating processes can be implemented on present day computers. The instructor should begin by considering the various implementations of hardware interrupts, by means of which peripheral devices interact with the central machine. If material is available, it is recommended that some attention is paid to queueing of channel commands and interrupt vectors [IBM/360, B6500, PDP-11].

When a process is interrupted, the status of this process should be stored in a control stack (or block) in order to make possible the subsequent resumption of this process. A good description is found in [36, Hansen]. When a central processor becomes available it is usually allocated to the highest priority process in the set of "unblocked" processes. (Note the distinction between "blocked" and "inactive.") The instructor could at this point introduce briefly the topic of processor scheduling, which is treated in detail in Module 7.

The primitive operations (P, V, LOCK, UNLOCK, BLOCK, WAKEUP) each require an implementation of a non-interruptable sequence of machine instructions, which can be achieved by masking off the interrupts. Booking a process in a waiting list and selecting a waiting process to wake up are part of the primitives; they should be implemented in a very simple form in order to minimize the execution time of the non-interruptable code. There have been several proposals for, and implementations of, less primitive operations than the ones mentioned above; examples are found in [7, Bernstein] and [8, Bétourné].

3.6 System Deadlocks

The discussion of deadlocks is in fact a continuation of the mutual exclusion and synchronization discussions, where it was stated that a process may have to wait until certain information becomes available. The system has maneuvered itself in a deadlock situation if none of the processes is going to provide the necessary information. It should be pointed out that a deadlock is caused by a conjunction of circumstances rather than by programming errors in the processes. Examples demonstrating this point are: circular waits, infinite repetition of request, two processes each holding half of a pair and asking for the other half.

The solutions of the deadlock problem should be classified in two kinds: 1) prevent its occurrence; 2) resolve the deadlock situation when it occurs. The first type of solution requires some knowledge in advance about the minimal needs of a process, but has the advantage that it does not restrict the number of working processes unnecessarily. Such a policy has been described in [35, Habermann]. The second type of solution does not demand any information about future behavior, but requires that preemption of resources or killing a process be allowed. An example of such a policy is [64, Murphy]. A good overall view is found in [11, Coffman].

At his option, the instructor may extend the discussions of this module by including a discussion of the overall structure of Operating Systems.

The ring structure of MULTICS [34, Graham] and the hierarchical level structure of the THE system [28, Dijkstra] define certain dominance relations between processes. Other hierarchical structures of processes are discussed in [24, Dennis & Van Horn]. OS/360 and the RC4000 system have these relations in the form of "parent-offspring." A well-considered structured set of dominance relations facilitates the designer to check the correctness of his design and allows him to apply the rule of "divide et impera" to it.

Module 3: Processes — Topic Outline

3.1 Parallelism in an Operating System

- Motivation of concurrent programming
- Motivation of asynchronous processes
- Abstract description of process
- Various aspects of process interaction

3.2 Mutual exclusion

- The problem of accessing shared data
- Critical sections and busy form of waiting
- Lock and unlock primitives; P and V operations

3.3. Synchronization

- Synchronization of events
- Block and wake up; the wake-up-waiting switch
- P and V operations used as synchronization primitives

3.4. Process communication

Sender-receiver relation
Generalization of sender-receiver concept
Some implementation aspects

3.5. Switching control

Hardware interrupt mechanisms
Process status and control block
Implementation of primitives

3.6. System deadlocks

General statement of the problem
How to resolve or prevent deadlocks
Dominance relations

Module 3: Processes — Reference List Guide

types: C — conceptual, D — descriptive, E — example, T — tutorial
level: S — student, A — advanced student, I — instructor

Key	Author	Type	Level	Importance
7	Bernstein et al.	E	S	4
8	Betourne et al.	E	S	3
11	Coffman et al.	T	A	2
13	Corbató and Vyssotsky	D	S	3
24	Dennis and Van Horn	C	I	3
25	Dennis et al.	T	I	2
26	Dijkstra	C	A	3
27	Dijkstra	T	A	1
28	Dijkstra	E	A	2
34	Graham	C	I	4
35	Habermann	C	A	3
36	Hansen	E	S	2
37	Hansen	D	S	3
41	Horning and Randell	C	I	4
49	Knuth	C	A	4
64	Murphy	C	A	4
72	Saltzer	C	S	2
76	Spier and Organick	D	S	3
86	Wirth	E	S	3

APPENDIX M3-A IMPLEMENTATION AND APPLICATION OF P AND V OPERATIONS

P and V operate on objects of type "semaphore." This type designates a data structure which is a pair (*s*, *Q*) in which *s* is counter variable and *Q* a set of "waiting processes." The conceptual difference of the P operation and other instructions in sequential processes is the fact that it may delay the execution of the next instruction. The operations of P and V are:

P (*sem*): decrement the counter *sem* and, if the result is negative, book the executing process on the waiting list *Q* and enter the wait state.

V (*sem*): increment the counter *sem* and, if the result is still not positive, select a process from the waiting list *Q*, remove it from *Q*, and release it from its wait state.

The P and V operations on a semaphore are "primitive" in the sense that their execution is uninterruptable by other P or V operations on that semaphore. Hazardous race conditions could arise if P and V operations could be broken apart in more primitive instructions, which could be executed in an arbitrary order by several processes. If, for instance, decrementing the semaphore is separated from the negative value test, two processes could find *sem* > O when *sem* = 1 and both would decrement sem, which would obviously have an undesired effect.

A reasonable implementation of P and V operations requires that, when it enters the wait state, a process releases resources that can be used effectively elsewhere. In particular, in a multiprogramming system, the central processor should be released by a process that enters its wait state through a P operation. Hence, a realistic implementation has this structure:

P (*sem*) executed by process *Y*:
  [*sem* : = *sem*-1;
  if *sem* < O
  then begin mark *Y* not ready;
      add *Y* to Q_{sem};
      go to RELEASE PROCESSOR;
  end]

V (*sem*): [*sem* : = *sem*+1;
  if *sem* ≤ O
  then begin *X* : = selection from Q_{sem};
      remove *X* from Q_{sem};
      mark *X* ready to run;
  end]

The brackets indicate that the enclosed actions are primitive (in order to avoid race conditions). On a machine with an interrupt system the P and V operations should be implemented as subroutines which are executed with all interrupts masked off (e.g., on an IBM/360 as SVC calls).

One can argue that the processor allocation, or the priorities of processes awaiting assignment to a processor, should be reconsidered when a V operation removes a process from a waiting list Q_{sem}, because indeed the set of processes ready to run is being expanded. In a system with only one central processor, however, it is not necessary to do so because the process that executed the V operation is still able to use the central processor effectively. The implementation allows a specific interpretation of the semaphore value: if positive, it indicates how many times a P operation will not cause a delay, whereas, if negative, it indicates the number of processes on the waiting list of this semaphore.

A hardware interrupt system is an implementation of P and V operations. Each interrupt handler routine can be regarded as a process which has performed a P operation when it enables the interrupt for which it is going to wait.

Although the corresponding V operation is in fact performed by the interrupt dispatching mechanism, one may regard the peripheral device or process that caused the interrupt as being the source of the V operation. In some implementations the occurrence of the V operation merely causes the appropriate interrupt handler process to be added to the queue of work for the central processor, the processor on which the interrupt occurred remaining under control of the process that was interrupted (MULTICS, THE). In other implementations the effect is an immediate allocation of the processor to the waiting process (TSS/360, PDP-10 Monitor); this is necessary in these systems in order to run the devices at maximum speed, because new commands may be presented to the devices only after an interrupt.

The interrupt mechanism is a more restrictive form of P and V operation, because it relies on the fact that Q_{sem} never contains more than one process, and moreover the identity of this process is associated uniquely with the particular semaphore (the interrupt bit). Therefore the interrupt dispatching mechanism can locate the waiting interrupt handler process simply by finding out what caused the interrupt.

The general structure of processes communicating via a communicating channel provides an excellent example of the application of P and V operations. A certain channel has a capacity of *C* messages. Senders S_i deposit messages and receivers R_i accept them. "Accept" and "deposit" are operations on the channel, each of which implies a sequence of operations on the channel status variables, of which the number of messages *M* and the number of empty slots *E*. In order to be able to interpret the channel status unambiguously at all times, accept and deposit should not be executed simultaneously. This is achieved by introducing a semaphore *mutex*, which has the initial value 1 and is used to realize the mutual exclusion of accept and deposit. Furthermore, Senders and Receivers should be synchronized with respect to the conditions "channel empty" and "channel full." When the channel is empty, the Receivers are to be blocked from attempting to accept messages and it is the Senders' obligation to notify the Receivers when a message is placed. Similarly, the Senders must be blocked from attempting to deposit messages when the channel is full and they should be notified when again there is an empty slot available.

The synchronization can be achieved by making *M* and *E* counting semaphores with initial value O and *C* respectively, and programming Senders and Receivers as follows:

S_i: **begin**	R_j: **begin**
prepare message;	P(*M*);
P(*E*);	P(*mutex*);
P(*mutex*);	accept;
deposit	V(*mutex*);
V(*mutex*);	V(*E*);
V(*M*);	process message
go to S_i	**go to** R_j
end	**end**

It is essential that the operations P(*E*) and P(*M*) are not executed between P(*mutex*) and V(*mutex*), whereas the order of the V operations could be reversed.

APPENDIX M3-B — SOME EXAMPLES OF EXERCISES IN CONCURRENT PROGRAMMING

1) Two cyclic sequential processes *A* and *B* intend occasionally to execute a "critical section" in their programs. A section of program is called "critical" because there is a requirement that one of them is allowed to enter its critical section only if the other is not passing its critical section at the same time; in other words, the critical sections must be mutually excluded. Program prologues and epilogues of the critical sections in *A* and *B* assuming that "copy the value of a variable" is the only uninterruptable action. [26, Dijkstra], [27, Dijkstra.]

2) Extend problem 1 to *n* processes $A_i, \ldots, A_n$, where $n \geq 3$.

3) *The Sleeping Barber* [27, Dijkstra.]
A certain Barbershop consists of two rooms: the waiting room *W* and the room *B* containing the barber chairs. The shop is so constructed that a sliding door *D* allows access either between *W* and *B*, or else between *W* and the street. Thus *D* allows either the barber to inspect the waiting room *W* or it allows a customer to come in from the street, but not both. If the barber inspects *W* and finds nobody there, he will return to *B* and fall asleep; otherwise he will invite the next customer to get his hair cut. If a customer enters the barbershop and he finds the barber asleep, he should wake up the barber. Program the barber and the customers using P and V operations.

4) If there is no restriction on how many customers may enter the waiting room, and if it is not known in which order the V operation is going to wake up the waiting processes, the customers of the previous problem could lock out the barber completely by denying him access to the door indefinitely. Modify the solution of the previous problem so that the barber cannot be locked out.

5) An Operating System contains a process *C* which Commands a line printer device *LP*, and a process *P* which deletes the line printer commands after completion. The processes and line printer *LP* communicate via an interrupt system with the following data structure:

  an "activation bit" *A*;
  a (hardware) semaphore *I*;
  a "switch bit" *S* and
  two command buffers *B*[0] and *B*[1].

Printer *LP* operates according to this algorithm:
  LP: **if** *A* = O **then go to** *LP* **else** *A* = O;
    execute *B[S]*;
    V(*I*);
    go to *LP*;

a. The first line is a P operation on the activation bit *A*. What is the body of the corresponding V operation (which ought to be performed by process *C*)?

b. Design programs for *C* and *P* (using additional variables and semaphores if necessary) such that the dual buffer system is utilized in the sense that *LP* may execute a command in one buffer while the other buffer is used to place or delete a command. Specify the initial values of the variables and semaphores assuming that the system starts with both the buffers being empty.

c. Try to combine *C* and *P* in one sequential program. Would it be more efficient to have one instead of two processes?

MODULE 4 – MEMORY MANAGEMENT

4.1 Introduction.

The study of modern memory systems concerns two distinct problems: 1) The set of techniques arising from our having to use two or more levels of memory in computer systems. These techniques encompass those of the "one-level store" or virtual memory. 2) The means of achieving systems supporting very general forms of modular programming; these include methods for dealing with objects whose size or structure may vary, and those allowing efficient sharing of procedure and data information among many processes. These methods involve enlarging the set of names (address space) within which a process may attempt to access procedure and data objects, either by a storage system separate from address space or by a structured address space. Although the second of these two aspects is the subject of Module 5, it is not completely independent of the first, which is the subject of Module 4.

4.2 Abstractions: Spaces and Mappings.

Most of the modern solutions to the automatic storage allocation problem derive from the *one-level store* introduced on the Atlas computer [47, Kilburn et al]. This machine distinguished "address" from "location," an address being the name for a word of information and a location being a physical site in which to store information. The set of all addresses (program addresses) a process can generate as it references information has come to be known as the *address space* of the process, and the set of all physical main memory location names (hardware addresses) has come to be known as *memory space*. By making this distinction, one is able to remove considerations of main memory management from the task of constructing a program, for the address space can be associated permanently with a program and can be made independent of assumptions about memory space. The memory management problem becomes the system's problem as it translates program addresses into location addresses during execution.

4.3. Motivation.

Computers have always included several distinct types of storage media, and the memory has been organized into at least two levels: main (directly addressable) memory and auxiliary (backup) memory. The desire for large amounts of storage has always forced a compromise between the quantity of (fast, expensive) main memory and (slow, cheap) auxiliary memory. The one-level store implements what appears to the programmer to be a very large main memory without a backing store. There are two ways of motivating it. The first starts from the idea of implementing a large, programmable virtual (simulated) memory on a machine having a relatively smaller main memory. The second traces the possible times at which program identifiers are "bound" to (associated with) physical locations. See [21, Denning, pp 143-157], [85, Wilkes, Ch 4].

The argument according to large virtual memory proceeds as follows. In the original stored program computers, main memory was quite small by today's standards; thus all programming was carried out in machine code and a large fraction of program development was devoted to the overlay problem, viz., deciding how and when to move information between main and auxiliary memory and inserting appropriate commands to do so into the program. When large (magnetic core) memories were introduced, pressure to solve the overlay problem was relieved. This respite was short-lived, however, for the introduction of algebraic source languages (e.g., FORTRAN and ALGOL) and the linking loader made it possible to construct large programs with relative ease. The resulting strain on main memory resources was aggravated because all the information involved in the execution of a computation was kept in main memory even though much of it remained unreferenced most of the time. Thus the need for executing large programs in small main memory spaces motivated hardware and software mechanisms that automatically moved information between main memory and auxiliary memory.

The argument according to binding time postponement proceeds as follows. There are five binding times of interest. 1) If the program is specified in machine language, addresses in it are bound to storage locations from the time the program is written. 2) If the program is specified in a high-level language, program identifiers are bound to storage locations by the compiler. 3) If the program is specified as a collection of subroutines, program identifiers are bound to storage locations by the loader. In 1-3, binding is permanent, once performed. It may, however, be also dynamic. 4) Storage is allocated (deallocated) to named objects on demand, such as at procedure activation (deactivation) time in ALGOL, and upon data structure creation (destruction) in LISP or PL/I. 5) Storage is allocated automatically by memory management hardware and software provided by the computer system itself. The sequence 1 through 5 is in fact the historical evolution of solutions to the storage allocation problem. It is characterized by *postponement of binding*. Although postponing binding time increases the cost of implementation, it also increases freedom and flexibility in allocating resources.

The two preceding paragraphs outline an approach by which the instructor can familiarize his students with the motivations, both qualitative and quantitative, for automatic storage allocation. Detailed accounts of the qualitative justifications can be found in [18, Denning], [21, Denning], [22, Dennis], and [70, Randell & Kuehner]. For quantitative justification, we recommend: 1) discussing Sayre's paper, which reports on comparisons between automatic and manual (programmer-controlled) storage allocation procedures [74, Sayre], also [21, Denning, pp 159-161]. and 2) assigning homework problems in which the student is asked to program with overlays devised by himself. An excellent example is a set of programs for multiplying two nxn matrices when main memory contains $3n^2$ n^2, and n data locations.

However one arrives at the conclusion that some form of dynamic storage allocation is required, one must also conclude that the programmer cannot handle it adequately himself: 1) he is not privy to enough information about machine operation to make allocation decisions efficiently. 2) solving the overlay problem at the programming level requires extensive outlays of a programmer's time, and the results are not consistently rewarding.

It is important to emphasize that there is a distinction between the *mechanisms* and the *policies* of storage management. The mechanisms are on a low level of abstraction in that they deal directly with the hardware features of the system, whereas the policies are on a higher level in that no detailed knowledge of machine organization is necessary or even relevant. The properties of one-level storage mechanisms are discussed in Sections 4.4-4.6, the policies in Section 4.7.

Examples of one-level store in contemporary systems are given in [6, Bensoussan et al], [70, Randell & Kuehner].

4.4 Formalization.

It is essential to study memory systems by means of the abstractions address space, memory space, and address map, and to emphasize them over and over. They are easy to grasp, and a student will be unable to comprehend the seemingly endless variety of existing memory systems unless he can be shown a pattern. As suggested earlier, memory management problems can be studied in terms of a mapping

$$f: N \rightarrow M$$

where N is the address space of a given program, M is the main memory space of the system, and f is the address map. If the word with program address x is stored at location y, then $f(x) = y$. If x is in auxiliary memory but not in main memory, $f(x)$ is undefined, and an attempt to reference such an x creates a fault condition which causes the system to interrupt the program's execution until x can be placed in memory and f updated. The physical interpretation of f is that a "mapping mechanism" is interposed between the processor and memory to translate process generated addresses (in N) into location addresses (in M). This reflects our earlier requirement that the address space N be independent of prior assumptions about M: since the programmer is aware only of N but not the two levels of memory, his program can generate addresses from N only.

4.5 Properties of One-Level Store.

The following paragraphs summarize six important properties of systems providing virtual memory.

Virtual memory fulfills a variety of programming objectives. 1) Memory management and overlays are of no concern to the programmer. 2) No prior assumptions about the memory space M need be made, and the address space N is invariant to assumptions about M. 3) Physically, M is a linear array of locations; typically, N is linear but contiguous program addresses need not be stored in contiguous locations since the address map f provides proper address translation. The address map thus provides,"artificial contiguity" and, hence, great flexibility when decisions about where information may be placed must be made (any unused location in M is suitable).

Virtual memory fulfills a variety of system design objectives arising in conjunction with multiprogramming and time-sharing systems: the abilities to 1) run a program partly loaded in main memory, 2) begin execution of a program shortly after it is presented, 3) reload parts of a program in parts of memory different from where they may previously have resided, and 4) vary the amount of storage used by a program.

Virtual memory provides a solution to the Relocation Problem. Since there is no prior relation between N and M, it is possible to load parts of N into M without regard to their order. It may still be necessary, however, to use a loader to link and relocate subroutines within N.

Speed-up. A two-level memory system with main memory M and auxiliary memory sufficiently large to contain all of N can normally be made to operate at nearly the speed of M, even though N may be very much larger than M. (To do this, it is necessary that a program's "working set" of information resides in M at each moment of time; see Section 4.7.) Two examples of this may be found in practice. 1) Main memory is magnetic core, auxiliary is drum, and the address map is implemented as a combination of hardware and software [21, Denning]. 2) Main memory is high speed semiconductor registers, auxiliary memory is magnetic core; according to this approach, known as "cache store" or "slave memory," the address map and management policies are implemented in hardware, the speeds of the two levels being so fast that decisions taken by software would be too slow [84, Wilkes].

Space-Time Trading. Let $T_e(n)$ denote the running time of the fastest program correctly solving a given problem when the address space size is n and the entire program is loaded in memory. The space-time tradeoff states that $T_e(n+1) \leq T_e(n)$. Suppose the memory size is m. The observed running time of this program is $T(n) = T_e(n), + T_O(n)$, where T_O is the overhead in the storage management mechanism; $T_O(n) = 0$ when $n \leq m$ and $T_O(n+1) \geq T_O(n)$ when $n \geq m$. Thus, when $n \geq m$ it may easily happen that $T(n+1) \geq T(n)$, i.e., space and time do not trade. Since m is assumed unknown a priori, the space-time tradeoff cannot be used while programming.

Protection. An elementary form of protection is provided: since a process may reference only the information in its address space, any other information is inaccessible to it.

4.6 Implementations.

Diagrams indicating the operation of address mapping mechanisms can be found in [21, Denning], [22, Dennis]. The address map f is normally represented in the form of a table, so that it may be accessed and updated efficiently. Two points are worth noting: 1) Since n (the size of N) is typically much larger than m (the size of M), a table with n entries, one for each element of N, is impractical. A table with m entries of the form $(x,f(x))$ is more viable. Since this table still must be indexed by a program address x, associative and hashing techniques may be used to access and update it. 2) When considering viable implementations of f, it is necessary to dispense with the notion of providing separate mapping information for each element of N. Instead, one partitions N into "blocks" of contiguous addresses and provides separate mapping information for blocks only. There are two alternatives: the block size is fixed and uniform, and the block size is variable. According to the former alternative—known as paging—blocks of address space are called "pages" and the blocks of main memory "page frames." The latter alternative arises when one considers defining the boundaries of block according to natural logical boundaries within the program, such as subroutines. For each alternative, one must consider: efficiency of the mapping mechanism, efficiency of storage utilization, and efficiency of possible allocation policies [21, Denning, pp 165-172], [50, Knuth, pp. 435-455]. The instructor should point out that there are two conflicting trends. On the one hand, fixed block size leads to simpler, more efficient storage management systems when properly designed, whereas on the other hand the need for efficient name management requires that *N* be partitioned into logical units called Segments [22, Dennis]. The value of "segmentation" (the ability to have Segments) is discussed in connection with dynamic and shared information structures as part of Module 5.

4.7. Policies.

A discussion of particular memory management policies and criteria for choosing one is appropriate at this point. The instructor should point out, however, that memory management is being studied now as a closed problem whereas it is in reality one component of a larger, systemwide resource allocation policy. The choice of a policy may, therefore, depend on additional issues to be raised in Module 7.

A memory management policy comprises three subpolicies [21, Denning, p 158]: 1) The fetch policy determines when a block should be moved from auxiliary to main memory, either on demand or in advance thereof. 2) The placement policy determines into which unallocated region of main memory an incoming block should be placed. 3) The replacement policy determines which blocks should be removed from main memory and returned to auxiliary. The complexity of the three subpolicies can be compared accordingly as block size is fixed or not; the complexity of a good policy may well affect the choice whether to use more than one block size. In demand paging systems, for example, all blocks are identical as far as the placement policy is concerned, so that memory management reduces to a study of replacement algorithms [4, Belady], [58, Mattson et al].

Demand paging is widely used and well documented in the literature, most of what is known being for the case of fixed main memory size. The instructor should give examples of policies and compare their relative performance [4, Belady], [21, Denning], [58, Mattson et al]. He should discuss the heuristic "principle of optimality" for minimizing the rate of page replacements: Replace the page with the longest expected time until reuse. This principle is equivalent to the "working set principle": A process should be run if and only if its working set is in main memory. A complete discussion of these points is given in [18, Denning], [21, Denning, pp 180-181].

The study of storage management policies may be rounded out by a treatment of policies for managing the auxiliary store. Considerations identical to those for managing main memory arise here: Whether block size should be fixed or not, what should be the block size(s), and how to store tables locating blocks in the auxiliary store. In addition, the use of "shortest access time first" disciplines for optimizing the performance of rotating-medium (disk, drum) request queues should be mentioned [1, Abate & Dubner], [21, Denning, pp 173-176].

Many a large paging system's auxiliary memory comprises drums and disks, the drums being used to swap pages of active programs to and from core, the disks being used for long-term storage of files. Files used by active programs may exist on both drum and disk simultaneously. The term "page migration" refers to the motion of pages of such files between drum and disk [85, Wilkes, p 45].

4.8. Extension to Multiprogramming.

As before, mechanisms can be treated separately from policies. In most systems a separate address space and address map are associated with each process. There are then, two ways to handle the mapping of the resulting collection of address spaces to the main memory. 1) Base and bound registers, or relocation registers, may be used to delineate a region of memory to be assigned to a given address space [21, Denning], [22, Dennis]. This is useful only when main memory can hold several address spaces (e.g. CDC 6600). 2) Main memory is treated as a pool of blocks, and the system draws from this pool to assign blocks to individual address maps as needed. The second alternative is more efficient to implement, especially if paging is used.

The working set principle is fundamental to multiprogrammed memory management. If this is not followed, attempted overcommitment of main memory can induce collapse of performance, known as thrashing [19, Denning], [21, Denning, pp 181-183].

Module 4: Memory Management — Topic Outline

4.1. The Needs of Modern Memory Systems

- Techniques for one-level store
- Techniques for name management

4.2. Introduction and Discussion of Abstractions

- Address and memory spaces
- Address map

4.3. Motivation

- Two level memory system
- Argument from large virtual main memory
- Argument from binding time postponement
- Quantitative justification
  - Empirical results
  - Homework problem demonstrating difficulty of overlays

4.4. Formalization of Concepts

4.5. Properties of One-Level Store

- Fulfilling programming objectives
- Fulfilling system objectives
- Relocation
- Speed-up; core-drum, cache-core
- Space-time trading
- Protection

4.6. Implementation

- Form and use of tables
- Fixed vs. variable block size

4.7. The Role of Allocation Policies

- Subpolicies: fetch, placement, replacement
- Tradeoffs: storage utilization, block size, policies
- Principles of demand paging
- Auxiliary memory management
- Paging drum
- Page migration

4.8. Extension to Multiprogramming

- Base-bound registers vs. pooling blocks
- Working set principle of management

Module 4: Memory Management — Reference List Guide

types: C — conceptual, D — descriptive, E — example, T — tutorial
level: S — student, A — advanced student, I — instructor

Key	Author	Type	Level	Importance
1	Abate and Dubner	C	A	3
4	Belady	DE	A	4
6	Bensoussan et al.	E	I	4
18	Denning	CT	S	1
19	Denning	C	I	4
21	Denning	CDT	S	1
22	Dennis	C	S	1
47	Kilburn et al.	CE	A	3
50	Knuth	C	S	2
58	Mattson et al.	C	A	3
70	Randell and Kuehner	DE	S	1
74	Sayre	E	S	2
84	Wilkes	C	A	2
85	Wilkes	CDT	S	2

MODULE 5 – NAME MANAGEMENT

5.1 Motivation

The techniques for memory management studied in Module 4 do not provide for the following important system objectives concerning the *computational memory*, i.e., that used to hold the procedures and data structures involved in computations:

1. Long-term storage of information.
2. Controlled sharing of access to data bases and procedures.
3. Creation, deletion, growth and shrinkage of information objects during the course of computations.
4. Program modularity, the ability of users to construct programs by linking together subprograms without knowledge of their internal operation.

These objectives must be met at the system level because they concern use of shared resources (space in main memory, peripheral storage devices and shared procedure or data bases). In each case questions of naming arise: objects of information (e.g., files) must be named for reference by computations; decisions to share objects and procedures should not result in conflicts in the meanings of names. As a result of this module, the student should understand how issues of naming objects arise; and he should learn the concepts and schemes through which the system objectives listed above can be achieved. The instructor should emphasize that there is as yet no single, generally accepted solution to the naming problem, a solution meeting all four objectives. Thus the instructor must concentrate on developing an appreciation of these issues, and the merits and limitations of known approaches to their resolution.

5.2 Basic Concepts

In a program, names are the symbols that specify the objects operated on by the program. In source language programs names are the identifiers of variables, structures, procedures and statements. When a program is compiled, most identifiers are replaced with numerical names (relative addresses) so that efficient accessing of instructions and data is possible: labels become addresses relative to the base of the machine code of the procedure, identifiers of local variables become addresses relative to the base of the procedure's activation record. Identifiers of external objects (e.g. other procedures and files) cannot be replaced by the compiler, and therefore must be retained in essentially unaltered form in the compiled procedure.

A compiled procedure is assigned a position in the address space of a computation (by a loader or a linking routine) so that it may be executed with other procedures during a computation. When this is done symbolic references between procedures are usually replaced with names in the form of addresses that locate the procedure within the address space of the computation.

Context

By nature of programmers and machines, the same name often will have two or more valid meanings. Some examples are: the same identifier may be used in distinct FORTRAN subroutines or ALGOL blocks; the address field of an instruction in the machine code for an ALGOL procedure must refer to different instances of variables for distinct activations of the procedure; machine language programs of different users may occupy the same memory locations having different meanings accordingly.

In each of these cases, the different meanings of a name are distinguished by additional information available to the compiler, loader, or hardware when the name is interpreted. This additional information is called the *context* in which the name is used. (Exercise: In each of the examples above, what are the contexts of the names?)

The context of a name need not be known at all stages of a name's use or transformation. For example, the compiler of a procedure cannot act on external names (those referencing other procedures, data structures, or files); it must leave such names in essentially the same form as they appeared in the source program. The context necessary for correct interpretation of these names is often not known until the procedure is assigned to the address space of a computation, or perhaps not even until execution is under way.

Distinct contexts for names used by different users are often provided by physically separating the information belonging to one from that belonging to another. This arrangement makes sharing of information (other than system information) difficult. If each user's information is catalogued in a "directory" but still is physically separated from other information, program modules can be shared only by the tedious and wasteful process of copying them from one directory to another.

A Fundamental Principle

In the discussions of dynamic structures and sharing of procedures to follow, there are illustrations of an important principle concerning the interpretation of names by a computer system:

The meaning of a name must not change during any interval within which independent procedures may use the name to refer to the same object.

This means, for example, that the positions of objects within the address space of a computation cannot be changed if these objects are referred to by independently specified procedures. The difficulties in using overlay schemes [22, Dennis], [54, Lanzano], [68, Pankhurst] for handling allocation of main memory stem from violation of this principle: because overlay schemes involve assigning two or more objects to overlapping areas in address space, the names (address) of such areas change in meaning during computation. To avoid chaos, each procedure making a change in the allocation of memory must inform all other procedures of the new arrangement of objects in address space. This requisite communication is, however, inconsistent with the objective that procedures be independently written.

5.3 File Systems

Computer systems generally provide for long term storage of information in the form of files. A file is an organized collection of data usually kept in peripheral storage devices such as magnetic drum or disk, or magnetic tape. The file management part of an operating system provides users means for generating and using files and manages the allocation of files to available space on storage units. Each user of the system is provided with a *directory*, in which his files are indexed or cataloged by names of his own choosing. In a number of systems, the objects indexed in a directory may include other directories as well as files, thus giving each user ability to create a directory tree, for organizing his collection of filed procedures and data bases in a hierarchy. A particular object is specified by a sequence of names, a *pathname*, that selects a path from a root of the directory hierarchy to the desired object. The file system provides also for protection and controlled sharing of files (this will be discussed fully in Module 6). These general concepts of file system organization are discussed in [16, Daley and Neuman], [24, Dennis and Van Horn] and [10, Clark]. Since the hierarchy of directories defines a mapping from pathnames to objects, the file system may be regarded as defining an address space for files. In most computer systems the address space defined by the file system is logically distinct from the address space for the computational memory. Hence, procedures and portions of files must be copied between the computational address space and the file address space during the course of a computation, an object being accessible only if it is in the computational address space.

Files themselves may be structured in several ways:

1. As a string of bits, characters or words.
2. As a sequence of records.
3. As an indexed collection of records, each record having a unique key. The records may be accessed by key or in the sequence defined by a natural ordering of the keys.

Files structured as ordered sets of words are often managed as sequences of blocks of fixed size for convenient allocation to free storage space (note the analogy with the use of paging to implement a linear address space in main memory). The block structure of files is a matter of implementation and is made invisible to user computations. The idea of "record" is an historically important way of delimiting fragments of data, originating in the use of punched cards. Files of records are stored on contiguous areas of storage devices (disk and tape) and are most suitable for sequential processing as is common practice in business applications. The indexed sequential file is important in systems that access data bases in "real time." Hash coding techniques are used to locate the record for an arbitrary key without searching the file [10, Clark], [42, IBM]. In discussing file structure, the instructor should distinguish carefully between structure that is seen by the user (the abstract structure of the file) and structure for implementation purposes that is hidden (or should be hidden) from the user.

Files may be of arbitrary size within wide limits, and may grow or shrink during processing; thus a file system provides facilities for manipulating dynamic structures. Modular programming may be done using program modules that obtain inputs from files and store results in other files. File directories provide long term storage for procedures and data and may include protection and shared access control features. Thus a general purpose file system would seem to achieve all four system objectives stated earlier. Yet, there are serious limitations.

I. A data file (or the portion of it being processed) must be copied into the computational address space to gain the advantage of accessing it through the hardware addressing facilities of the computer. The implied loss of efficiency will be severe for computer limited computations if files are used as the basic objects.

II. Procedures or program modules retrieved from the file system must be loaded into the computational address space prior to execution. In most systems all procedures must be loaded in advance of execution, wasting address space and making it impossible to activate procedures whose names are not known until retrieved from data bases or typed in from a terminal. If procedures may be loaded dynamically, the copying and linking steps will be time consuming.

III. There are few generally accepted standards for the structure and naming of files and for the primitive file operations implemented by file systems. Existing standards relate to COBOL, a language for data processing applications. Conventional file systems are not a suitable base for efficient implementation of procedures expressed in ALGOL, FORTRAN or or PL/I.

IV. Clashes of identifiers (file names) appearing in independently written procedures are not avoided. This matter is discussed further in the Section 5.6, on modularity.

5.4 Segmented Address Space

These four limitations result at least in part from the distinction between the computational memory and the file memory, and can be relieved by using the virtual memory concept to remove this distinction. This has been achieved by combining use of file directories with a large, *segmented address space*. The address space is divided into a large number of *segments*, each being potentially large enough to hold any object (procedure or data file) indexed in the file directories. Reference by a computation to information in the address space is made by a pair of values (segment number, word number), segment numbers being assigned to procedures and data files when they first are referenced by a computation. After the first reference, the given procedure or data object is bound to a particular segment of address space and, thereafter, the object may be referenced efficiently as a resident of address space rather than by a search of its pathname in the directory hierarchy.

Two methods are in use for implementing a segmented address space. In one [21, Denning], [43, Iliffe], [44, Iliffe and

Jodeit], [70, Randell and Kuehner], the segment number is used as an index in a system-managed table of "descriptors" or "codewords." A descriptor (codeword) locates the origin of a segment within the main memory if space in main memory has been allocated for the segment, or in peripheral memory otherwise. The other method divides segments (linear name spaces in their own right) into pages and uses two levels of system tables to map (segment number, word number) pairs into main memory locations [21, Denning], [22, Dennis], [15, Daley and Dennis]. In both schemes the system tables also contain access control and protection tags.

The use of a segmented address space is valuable for providing independently written procedures space for large dynamic structures, and for permitting all objects to be shared by computations if desired. These ideas are examined in the following paragraphs.

In current systems, the advantages of segmented address spaces have not compensated for the difficulty and complexity of their efficient implementation. For example, the mechanisms required for linking procedures together in the address space of a computation are intricate [15, Daley and Dennis] and should be considered an advanced topic.

5.5 Dynamic Structures

A dynamic data structure is an organized collection of information that changes in extent during a computation. Two types of such data structures are in common use: 1) variable-size tables, such as symbol tables, stacks, or matrices; and 2) linked-list structures [31, Foster]. Both types (or combinations thereof) require some mechanism for managing the address space in which they reside. With respect to the first type, there are two approaches, depending on the size of the address space and the nature of the mapping to memory locations if addresses are virtual. If the address space is sufficiently large, each structure may be assigned to a separate segment of address space, large enough so the structure may grow and contract without conflicting with other structures. Since large parts of the address space will be unoccupied, this approach is of interest only when a virtual memory mechanism is present to map the occupied parts of address space into memory (e.g. a segmented address space).

If, on the other hand, a large address space is not available but the structures involved in a computation will fit into memory space, a system of routines may be provided for managing the assignment of parts of structures in the available space. Schemes for dynamically allocating contiguous blocks within a relatively small address space are described in [50, Knuth, pp 435-455], and are similar in function to the routines discussed below for managing linked structures.

A linked list structure is a collection of items, each consisting of a datum and a pointer (or pointers) to other items. The pointers are addresses that locate items within the address space [31, Foster]. A particular datum is accessed by following a chain of pointers from a single item that serves as the root of the structure. The system routines which access the structure on behalf of a program have three functions: 1) Free storage-management, i.e., handling allocation of new items, deletion of old items, and maintaining records of free space; 2) garbage collection, i.e., identifying items which have been deleted but not yet returned to the pool of free items; and 3) compaction, i.e., the relocation of live items in the address space so that the live items occupy a contiguous region [45, Jodeit], [50, Knuth, pp 435-455.]

Compaction is used when the structures must be placed into as small a portion of address space as possible. Since the compaction process invalidates the addresses in the items until it is completed, no accesses can be permitted to the structure during compaction.

This is an important illustration of the general principle stated at the beginning of this module. Compaction changes the names (addresses in this case) by which components of data structures may be referenced by computations. If two computations access the data structures concurrently, both must be halted during compaction and, moreover, any addresses (pointers to data structure items) held in private working storage of either computation must be corrected. This is why compaction is avoided in the design of storage allocation schemes for multiprogram operating systems. Compaction is often used to limit linked structures to a smaller contiguous portion of virtual address space to improve performance of single process computations on a paged computer.

Linked list structures are the standard representation of data in certain programming languages such as LISP; but they are often useful in providing efficient storage of complex structures for any application. For this reason, facilities for manipulating linked structures are provided in languages such as PL/I and ALGOL 68.

5.6 Modularity

The construction of a computer program can be greatly simplified if its major parts are already in the form of program modules that can be easily combined without knowledge of their internal operation. The student should learn the characteristics of computer hardware and software essential to modular programming, and understand how practical systems achieve or fail to achieve these necessary properties.

Two fundamental requirements for successful modular construction of programs are:

1. Program modules to be used together must employ consistent representations for all information exchanged among them.
2. A universal scheme must be established by convention for interfacing program modules with one another.

Modularity can be achieved for a particular group of system users if they adopt uniform conventions among themselves for data representation and intermodule communication. Such standards can be developed and agreed upon for any computer system, but seldom without some compromise between degree of generality and efficiency of program execution. Modularity achieved through use of shared files in the manner described earlier is an example. Different user groups, however, are likely to adopt conflicting conventions, and programs which do not honor such conventions are unusable as modules.

This discussion of the limitations of imposing conventions on existing systems should be followed by study of system characteristics that permit any program written for execution

by the system to be used as a module in the construction of larger programs.

The requirement for consistent representation of intercommunicated data is met if all modules are expressed in the same source language, processed by the same compiler, and use only the data types provided by the language. Otherwise, this requirement cannot be satisfied without extreme care in the design and implementation of the language processors and execution environment [80, Wegner], [59, McCarthy et al].

The most important form of program module is the procedure, for which basic implementation concepts were treated in Module 2. For modular programming, a procedure's author must be free to choose whatever names he desires for objects referenced by the procedure—instructions, variables, data, structures, and other procedures—without clashing with independent choices made by the authors of other procedures. To aid in understanding the solution of this and related naming problems, the notions of "argument structure" and "procedure structure" may be introduced.

The information which does not vary from one activation to another of a procedure P is called the *procedure structure* of P. It consists of

1. The code (machine language) of procedure P.
2. The procedure structures of other procedures that are used by P in the same way in every activation of P.
3. Any data structures (e.g., *own* data in ALGOL 60, or STATIC data in PL/I) that "belong" to the procedure in the sense that all activations of P refer to the same instances of the structure.

The parts of the procedure structure must be referenced directly by names appearing in the code of the procedure. These names are chosen by the author of the procedure and should be of no concern to the user of the procedure; the context in which these names are interpreted must therefore be distinct for each distinct procedure.

The information which may change from one activation to another consists of

1. The input data.
2. The output data.
3. The activation record (working storage).

The *argument structure* consists of the input and output data. Conceptually, the components (simple or compound) of the argument structure can be assumed numbered by distinct integers 1,2,3, ... In his coding of the procedure, the author may associate symbolic names $x_1 x_2 x_3, \ldots$ with these numbered components. Similarly, the user of a procedure may associate his own symbolic names $y_1, y_2, y_3, \ldots$ with these numbered components. Thus the ordering and structure of the components is fixed and part of the interface specification, but both author and user are free to choose names for them as they please.

The names used in a procedure to reference its activation record are local and the procedure's author must be free to choose them as he pleases. Since these names refer to different data in different activations, the context in which these names are interpreted must be different for each activation. Furthermore, the working storage must be able to expand to meet the storage requirements of the activation, which may be arbitrarily large.

If one procedure may be called from several independently written procedures, nonlocal references (as in ALGOL 60) have no meaningful interpretation. Also, external references (as in a FORTRAN implementation) only make sense in a global context, where name clashes are possible. Thus modular programming must be done *without the aid of side effects* —all input and output data of a program module must be conveyed as components of the argument structure.

Implementations of ALGOL 60 provide distinct working storage areas for nested procedure activations and therefore handle recursive programs, the amount of working storage being specified upon procedure activation. Thus limitation II and, to some degree limitation I, are overcome. In systems that offer ALGOL 68 or PL/I, means for representing and manipulating linked structures are provided, thereby removing limitation III for an important class of data structures. Yet clashes between names (of external procedures and files, for example) are still possible and limit the degree to which modular programming is possible. This problem is discussed in [24, Dennis and Van Horn, pp 151-154], [78, Vanderbilt, Chapters 2 and 3].

Clashes could be avoided by providing two contexts for the interpretation of "external" names occurring within procedures. Context for procedures and files that are part of the procedure structure would be provided by a *procedure directory* associated with the procedure in execution. Context for procedures and files named in the argument structure would be provided by an *argument directory* selected by the calling procedure. In this way, all names would be interpreted in appropriate contexts, and all possibilities of name clashes avoided. Although this idea is not implemented in any current system, some similar scheme will be necessary in future systems if modular programming is to be achieved.

With these concepts as background, the class can study the extent to which modular programming is permitted by various classes of systems. In the FORTRAN environment of Module 1, each subprogram has a single, permanently-assigned, fixed activation record. Context for naming the parameters of a call is typically provided by the return address, a list of parameters being in some fixed location with respect to each point of call. Internal references within a subroutine are assigned meaning within the body of the subroutine only. Some limitations of this are

1. Working storage is not expandable.
2. Recursion is not implemented.
3. There is no means for representing dynamic objects.
4. External references made by subprograms are interpreted within the (global) context of the loader's symbol table; thus a name clash will occur if two subprograms use the same name to refer to distinct objects. (This can be especially troublesome if subprograms written by authors for their personal environments are used in other environments.)

5.7 Sharing (Advanced Topic)

To permit sharing of procedure and data among users of a computer system, it is necessary to have a systemwide naming scheme so that one user can reference objects owned by others. One method for doing this uses a systemwide directory tree and allows directories to contain links to other directories [16, Daley and Neumann].

As an advanced topic, the instructor may discuss implementations in which a single copy of a procedure or data object in main memory is used by all computations sharing access to the object. There are three motivations for doing this: 1) conserve main memory, 2) avoid redundant copies of information, and 3) reduce overhead required to move extra copies in and out of main memory.

In most systems sharing of code in main memory is limited to supervisor and library programs; it is implemented by reserving for these shared objects a certain known portion of each user's address space.

If, on the other hand, every procedure in a computer system must be regarded as potentially shared among computations, the implications for addressing mechanisms of the system are far-reaching [22, Dennis]. Not only is it necessary to separate data and procedure information by means of "pure procedures," but a satisfactory means of transferring control between procedures is needed [15, Daley and Dennis].

If control transfer instructions contain absolute addresses in the address space of a computation, a shared procedure must be assigned to the *same* position in the address space of every computation that uses it. It follows that each address space must sufficiently large to hold *all* potentially shareable procedures. Since there is no way for the system to know which procedures users may wish to share, it is attractive to implement a single address space for all computations in the system. No such system has been constructed, and it remains to see whether such an implementation would be practical.

Now, suppose control transfer instructions contain addresses relative to the base address of the procedure in which they appear. A procedure may now be assigned different positions in the address spaces of different computations, but the processor must contain a base register containing the base address of the procedure in the applicable address space; all control transfer instructions are interpreted relative to this base register. Provision must be made for reloading the base register whenever control is passed between procedures, and a scheme for properly implementing external references must be worked out. One complete scheme is described in [15, Daley and Dennis]. Because addresses have different meanings for different computations, this scheme has some serious disadvantages. Communication between computations is difficult, since directory names rather than addresses must be used to identify objects in messages. Also, once a procedure has been assigned a position in address space of a computation, it is not legitimate for the system to delete it until the user can guarantee no further attempt to access it by its assigned address will be made. As a result, occupancy of address space will tend to increase as a computation references new objects; thus a program that runs for an extended period, continually accessing new information, will have to manage its own use of address space.

Module 5: Name Management — Topic Outline

5.1 Motivation

System objectives concerning storage and access to procedures and data bases.
Issues concerning treatment of names by system
Objective to appreciate issues and understand merits of known techniques

5.2 Basic Concepts

Forms of names: identifiers, addresses; translation of names by compiler, loader, and system.
Context for interpretation of names; examples
A fundamental principle: meaning of name must not change while in use by independent procedures
Example: overlay schemes.

5.3 File Systems

Files, directory hierarchies
Structure of files, their representation on strange devices, implementation
Achieving system objectives by use of files to represent data
Limitation of file systems

5.4 Segmented Address Space

Segments, two-component addresses, binding procedure or data to address space
Implementations
- with base registers
- with paging

5.5 Dynamic Structures

Two types: arrays of variable size; linked structures
Management of address space for dynamic structures: free space management, garbage collection
Compaction, a violation of the naming principle
Desirability of large segmented address space.

5.6 Modularity

Fundamental requirements
- consistent data representations
- interfacing conventions

Concepts for modular use of procedures
- procedure structure, argument structure

Limitation of systems for modular programming
- FORTRAN
- ALGOL 60
- ALGOL 68 or PL/I

5.7 Sharing

Use of links in directories for permitting controlled access
Motivations for shared use of information in main memory
Implementation alternatives
- common address space for all computations
- distinct address spaces with relative addressing; the problem of linking.

Module 5: Name Management — Reference List Guide

types: C — conceptual, D — descriptive, E — example, T — tutorial
level: S — student, A — advanced student, I — instructor

Key	Author	Type	Level	Importance
10	Clark	D	I	4
15	Daley and Dennis	CD	A	2
16	Daley and Neumann	CD	S	2
21	Denning	CDT	S	2
22	Dennis	C	S	1
24	Dennis and Van Horn	CD	S	1
31	Foster	T	S	2
42	IBM	D	I	4
43	Iliffe	CD	S	1
44	Iliffe and Jodeit	CD	S	1
45	Jodeit	D	I	2
50	Knuth	DET	S	2
54	Lanzano	DE	I	2
59	McCarthy et al.	CD	S	2
68	Pankhurst	D	S	2
70	Randell and Kuehner	D	S	3
72	Saltzer	C	A	3
80	Wegner	CD	S	3

"Protection" is a general term for all the mechanisms which control the access of a process to other objects in the system. There is an immense variety of implementations, it being normal for a single system to have several different and unrelated protection mechanisms; for example, supervisor/user modes, memory relocation and bounds registers, a numbering system for open files, access control by user to file directories, and a password scheme for user identification.

The study of protection has been deferred until this module of the course because the concepts studied so far could be embodied in a computer system having no protection features, it being assumed that all users are friendly and infallible. This observation should not be taken to imply that protection is a minor consideration. Indeed, real users–and the programs they write–are far from infallible, and privacy is a central issue in most systems. As a result, protection considerations are in fact pervasive in system designs.

The abstractions developed in this module should be illustrated from the operating systems being studied as examples. The material in this module is presented here in considerable detail because the literature is inadequate.

6.1 Motivation

The original motivation for putting protection mechanisms into computer systems was keeping one user's malice or error from harming other users. A user can harm others in several ways:

1. By destroying or modifying another user's data;
2. By reading or copying another user's data without permission;
3. By degrading the service another user receives, e.g. using up all the disk space or getting more than a fair share of the processing time. An extreme case is a malicious act or accident which crashes the system (the ultimate degradation).

More recently it has been realized that these reasons for wanting protection are just as strong if applied to "programs" as well as "users." This line of reasoning leads in three directions:

1. Toward enforcing the rules of modular programming so that it is possible to guarantee (through the protection system) that errors in one module will not affect another one. This kind of control engenders confidence in the reliability of a large system, since the protection provides "fire walls" which prevent the spread of trouble [34, Graham], [52, Lampson].
2. Toward the support of proprietary programs, so that a user can buy a service in the form of a program which he can only call, but not read [52, Lampson]. A simple case is a proprietary FORTRAN compiler whose use is charged by number of statements compiled. A more complex case is a proprietary program which compares trial data against a proprietary data base.
3. A third case may suggest that some generality is really required to handle those problems, rather than a few ad hoc mechanisms. This is the construction of a routine to assist in the debugging of other programs. A debugger needs to be able to access all the objects the program being debugged can, but must protect itself and its data (breakpoints, symbol tables, etc.) from destruction by malfunctions in the program being debugged [51, Lampson], [52, Lampson].

6.2 Protection Domains

At the foundation of any protection system is the idea of protection environments or contexts. Depending on the context in which a process finds itself, it has certain powers; different contexts have different powers. A simple example of a two-context system, the contexts being implemented in hardware, is computer with supervisor and user (problem) states (modes). A program executing in supervisor state can execute I/O instructions, set the memory protection registers, halt the machine, etc., but it may do none of these things in the user state. A somewhat more elaborate example is OS/360 MVT, in which there is one supervisor context and up to 15 user contexts; the limit of 15 is enforced by the use of 4-bit keys in the 360's memory protection system. Yet another example is the individual users of a multiaccess system–each has his own protected program and files, so that there are at least as many protection contexts as there are users [85, Wilkes].

These examples suggest the generality and subtlety of the idea. Many words have attempted to capture it: protection context, environment, state, or sphere [24, Dennis and Van Horn], capability list [53, Lampson], ring [34, Graham], domain [52, Lampson]. The word "domain" will be used here, since it is somewhat neutral and has fewer misleading associations than the alternatives. An idealized system is described below in order to clarify the meaning of the term "domain" and provide a framework for the description of real systems.

The idealized system consists of processes which share nothing and communicate with each other only by means of *messages*. A message consists of an identification of the sending process followed by an arbitrary amount of data. The identification is supplied by the system and therefore cannot be forged. Processes are assigned unique integer names by the system. Any process may send messages (at its expense) to any other process. Messages are received one at a time in the order in which they were sent. See [37, Hansen] for an actual system very similar to this one, but described from a somewhat different viewpoint.

Within this system every object belongs to exactly one process and cannot be accessed by any process other than its owner. Each process therefore defines a single domain. It may also be regarded as a separate machine, complete with memory, file storage, tape units, etc., and isolated from all contact with other processes except for the message transmission facility. This scheme constitutes a logically complete (though inefficient) protection system, except for two points which are discussed later.

The following point (which has nothing to do with protection) is important. With this system we can simulate a subroutine mechanism, regarding one process *(A)* as the calling routine and another *(B)* as the routine being called. To call *B*, *A* sends *B* a message specifying the parameters and then waits for *B* to reply. To return, *B* replies with another message containing the value, if any, and then begins waiting for another call.

Unlike an ordinary subroutine calling mechanism, this one works even if *B* must be protected from *A*, e.g. if *B* is the supervisor and *A* a user program. It works because *B* determines where he is "entered," namely at the point where he waits for *A*'s message. Random transfers of control to an arbitrary point in *B* are not possible. (Multiple entry points are possible, since *B* can decode one of the parameters to select an entry point.)

Furthermore, the "return" is protected also. Thus, if *A* mistrusts *B*, e.g. in the case of a command processor calling a user program, the same argument shows that *B* will not be able to return to *A* except in the manner intended by *A* Spurious additional "returns" (extra messages) from *B* are impossible as well, since *A* knows when he is expecting a return message from *B* and can ignore messages at other times. The scheme clearly works even if each domain mistrusts the other, as in the case of calling a proprietary program [53, Lampson].

What if *A* calls *B* by this mechanism and *B* never returns, because *B* is faulty or even malicious? If *A* wishes to guard against this possibility, he need only arrange (before calling *B*), to receive a message from some reliable system process *C* after an elapsed time longer than *B* is expected to run. If the message *A* receives next is from *C* rather than from *B*, then *A* knows something has gone wrong and can proceed to take corrective action.

Finally, we show that domains are protected against unauthorized call. Recall, that, as part of each message, the system supplies the identity (name) of the caller. This identification may be thought of as a signature, a seal, a badge, or a ticket, which *B* can use to check the validity of the call. The key point is: *the identification is supplied by the system which guarantees that it cannot be forged*. This point is so simple and yet so subtle that we will illustrate it with an example. Suppose that *A*, whose system identification is 6389, sends to *B* a message consisting of three numbers: 31, 45, 9. What *B* will receive is four numbers: 6389, 31, 45, 9. The first number is attached to the message by the system from its knowledge of the sender's identity. There is no way for *A* to affect the value of the first number in the message. From *B*'s point of view, then, the message starts with a single identifying integer. If *B* is expecting a message from *A*, all he must do is look through his message buffer until he finds one starting with *A*'s identification number. How *B* gets to know *A*'s name is an interesting question which will be examined below, but the following simple scheme will suffice: *A*'s user at his terminal asks *A* for the number and shouts it across the room to *B*'s user, who types it in to *B*. Remember that this number is *not* a password. Knowing it allows *B* to give *A* access, but does not help anyone else (including *B*) to impersonate *A*, as the description of message handling given above should make perfectly clear.

The kind of protection or access control which can be enforced with this system is extremely flexible and general, since arbitrary programs can be written by users to make the protection decisions. (Suggested exercise: show how an instructor could implement a grading program which gives student programs at most three tries at obtaining a correct answer.)

As was suggested earlier, the system we have been describing has two major flaws. First, it is impossible to regain control over a runaway process, since there is no way to force a process to do anything or to stop it. This makes debugging difficult. Although such a process cannot do any damage, it can waste resources. Second, an elaborate system of conventions is required to get processes to cooperate. Suppose, for example, that process *A* has some data which it wants to share with processes or friends of *A*'s owner. It is necessary for *A*'s owner, whom we regard as another process communicating with *A* via a terminal, to learn the names of his friends' processes and to include in *A* some subroutine which knows these names and responds appropriately to messages carrying them as identification. Moreover, *A* and its correspondents must agree on the interpretation of messages.

The protection system we have described is as devoid of convenience as is a central processor without an assembler. Just as the processor needs an elaborate system of conventions in the form of loaders, binary formats, assemblers, etc., to make it usable, so a protection mechanism requires a system of conventions on process names, data formats, etc. The issues raised by these two points are discussed in the next section.

6.3 Objects and Access Matrices

In order to provide facilities for external control of processes, it is necessary that the protection facility allow for controlled access of one domain by others. (The simple scheme described above allowed no access at all.) Thus there must be a way of describing what is to be shared and how access is to be controlled among domains. Access to processes can be controlled by a simple tree structure [37, Hansen], [51, Lampson], but it can also be handled more generally by the same machinery which we will introduce below. (It is not at all clear that the scheme described below is the only, or even the best, set of conventions to impose, but it does have the property that most of the schemes used in existing systems are special cases of this one.)

The more general protection system can be described in terms of another idealized system with three major components: a set *X* of *objects*, a set *D* of *domains*, and an *access matrix* (access function) *A*. Objects are the things in the system which have to be protected. Typical objects in existing systems are processes, domains, files, segments, and terminals. The question what to designate as objects is a matter of convention, to be determined by the protection requirements of each system.

Objects have names with global validity, which we will think of as 64-bit integers. Object names are handed out by the protection system on demand, and their interpretation is up to the programs which operate on the objects. This point point is clarified with an example below.) The object names do not have to be 64 bits; there should, however, be an extremely large set of potential object names, for two reasons.
1) Each object must have a unique global name, and an

object's name may not be reused after the object ceases to exist. 2) If a system error changes a bit in a name, the resulting bit string should be another valid name only with exceedingly small probability. The MULTICS system, for example, uses the time in microseconds since 1900 as the unique object name.

As before, a domain is a protection context; domains are the entities which have access to objects. Each domain has potentially different access to objects than other domains. In the system of Section 6.2 each domain was defined by a process, and had exclusive access to its own objects and none to any others. This idea is now being generalized so that objects can be shared among domains. There are two ways to view this generalization:

1. Each domain owning objects in the system of Section 6.2 agrees by convention that it will do certain things with these objects upon demand from some other domain, provided the other domain has access according to the rules below.
2. For certain "built-in" objects, at least, the access rules below will be enforced by mechanisms already present in the system for other reasons (whether this is hardware, as in the case of memory protection, or software, as in the case of file directories, is not important). This may lead to greater efficiency (memory protection is an extreme example) but it is not general and must be supplemented by conventions as in point 1 above if the system is to be extensible. As far as the protection system is concerned, it makes no difference whether we assume the existence of such other mechanisms or not.

Note that domains are objects, not sets. In particular, objects do not "belong to" domains.

The access of domains to objects is defined by the access matrix *A*. Its rows are labeled by domain names and its columns by object names. Element *A[i,j]* specifies the access which domain *i* has to object *j*. Each element consists of a set of strings called *access attributes*; typical attributes are "read," "write," "wakeup." We say that a domain *i* has "*x*" access to an object *j* if "*x*" is one of the attributes listed in *A[i,j]*. Associated with each attribute is a bit called the *copy flag* which controls the transfer of that access attribute in a way described below. With the access matrix of the figure, for example, domain 1 has "owner" access to file 1 as well as explicit "read" and "write" access. It has given "read" access to this file to domains 2 and 3.

	Domain 1	Domain 2	Domain 3	File 1	File 2	Process 1
Domain 1	*owner control	*owner control	*call	*owner *read *write		
Domain 2			call	*read		wakeup
Domain 3			owner control	read	*owner	

**copy flag set*

Figure: Portion of an access matrix

Entries in the access matrix are made and deleted according to certain rules. The following are examples of such rules. *A* domain *d* can modify the list of access attributes for domain *d'* and object *x* as follows (examples assume the access matrix of the figure):

1. *d* can remove access attributes from *A[d',x]* if it has "control" access to *d'*. Example: domain 1 can remove attributes from rows 3 and 3.
2. *d* can copy to *A [d',x]* any access attributes it has for *x* which have the copy flag set, and can say whether the copies attribute shall have the copy flag set or not. Example: domain 1 can copy "write" to *A* [domain 2, file 1].
3. *d* can add any access attributes to *A[d',x]*, with or without the copy flag if it has "owner" access to *x*. Example: domain 1 can add "write" to *A* [domain 2, file 2].

The reason for the copy flag is that without it a domain cannot prevent an undebugged subordinate domain from wantonly giving away access to objects.

The rules above do not permit the "owner" of an object to take away access to that object. Whether this should be permitted is an unresolved issue. It is permitted by most systems; see [78, Vanderbilt] for a contrary view, according to which an "owner" has in essence entered a contractual agreement to provide services to other domains. If this view is adopted, the following rule might be appropriate.

4. *d* can remove access attributes from *A [d',x]* if *d* has "owner" access to *x*,,provided *d'* does not have "protected" access to *x*.

The "protected" restriction allows one "owner" to defend his access from the other "owners." Its most important application is to prevent a program being debugged from taking away the debugger's access; it may be very inconvenient to do this by denying the program being debugged "owner" access to itself.

The protection system itself attaches no significance to any access attributes except "owner," "control," and "protected." Thus the relationship between, say, the file-handling module and the system is something like this. A user calls on the file-handler to create a file. The file-handler asks the system for a new object name *n*, which the system delivers from its stock of 2^{64} object names. The system gives the file-handler "owner" access to object *n*. The file-handler enters *n* in its own private tables, together with other information about the file which may be relevant (e.g. its disk address). It also gives its caller "owner" access to *n* and returns *n* to the caller as the name of the created file. Later, when some domain *d* tries to read from file *n*, the file-handler will examine *A [d,n]* to see if "read" is one of the attributes, and refuse to do the read if it is not.

6.4 Some Implementation Techniques

Since *A* is sparse, it is not practical to store it in the obvious way. The most intuitively simple alternative would be a global table *T* of triples *(d,x,A[d,x])* which is search whenever the value of *A [d,x]* is required. Unfortunately, this usually is impractical:

1. Memory protection is almost certainly provided by hardware which does not use *T*. This is the major area in which the operating system designer has little control. (It is discussed in Section 6.5.)
2. It may be inconvenient to keep all of *T* in fast-access memory, since at any given time most objects and perhaps most domains will be inactive. An implementation is therefore needed which keeps only currently relevant parts of *A* readily available in fast-access memory.
3. Objects or domains may happen to be grouped in such a way that *T* is very wasteful of storage. A simple example is a public file, which would require an entry in *T* for every domain.
4. It may be necessary to be able to obtain a list of the objects which a given domain *d* can access, or at least the objects for which *d* is responsible or is paying for.

An implementation which solves 2 and 4 directly attaches to each domain *d* a table of pairs *(x,A[d,x])*. Each of these pairs is often called a *capability* [24, Dennis and Van Horn], [43, Iliffe], [52, Lampson], [53, Lampson], [85, Wilkes]. If the hardware provides for read-only arrays which can only be generated by the supervisor, then each capability can be implemented as such an array, containing the name of the object, and a suitable representation of the access attributes (perhaps as bit strings). Most hardware does not provide the kind of protected arrays we have been assuming, but they can easily be simulated by the supervisor (at some cost in convenience) on any machine with some kind of memory protection. It is usually convenient to group capabilities together into capability lists or *C-lists*. A domain is then defined by a C-list (and its memory, if that requires special handling; see Section 6.5).

With this kind of implementation it may be convenient to allow additional information to be stored in a capability, e.g. the disk address of a file, or a pointer to some table entry to save the cost of looking up the object name [53, Lampson]. (Exercise: devise a mechanism for controlling who gets to alter this additional information.)

Capabilities can also be used to solve problem 3 above. All we have to do is construct a tree of domains, each with a set of capabilities or C-list [24, Dennis and Van Horn], [43, Iliffe], [78, Vanderbilt]. Everything we know about tree-structured naming schemes can then be applied to economize on storage or share capabilities.

A completely different approach to storing *A* attaches the protection information to the object *x* rather than the domain, in the form of a list of pairs *(d,A [d,x])*. With each object *x* there will be a procedure A_x *(d)* which returns *A [d,x]*. The procedure is provided by the owner of the object and can keep its own data structures to decide who should get access. Note that at least some of these procedures will have to refrain from accessing any other objects in order to prevent infinite recursion. This is the idea of an *access control list*, such as is used in MULTICS.

As before, it is essential to note that the procedure A_x gets a domain name as argument, and this cannot be forged (see Section 6.2). Unique names, however, may not be convenient for the procedure to remember; access is likely to be associated with a person or group of people, or perhaps with a program. For example, capabilities can be used as identification, since they have the essential property that they cannot be forged. We will call a capability used for identification an *access key*; it is a generalization of a domain name [52, Lampson]. Then all the access control procedure A_x needs to know is what access keys to recognize. Each user (indeed, each entity which needs to be identified by an access control procedure) obtains a unique access key from the supervisor, records it, and transmits it to those who wish to grant him access. They then program their access control procedures to return the desired attributes when that key is presented as an argument.

In order to avoid the inconvenience of arbitrary access control procedures, one may attach to each object an *access lock list* consisting of pairs (key value, access attributes). It works in the obvious way: if the value of the key presented matches the value in one of the locks, the corresponding attribute is returned. Alternatively, one may regard this scheme as a generalization of one of the first protection systems, that of CTSS, which, instead of a key value, employed the name of the user as identified at login [14, Crisman], [52, Lampson], [85, Wilkes].

One access list per object is likely to be cumbersome. Many systems group objects into *directories*, which in turn are objects, so that a tree structure can be built up. This adds nothing new, except that it introduces another kind of tree-structured naming [85, Wilkes]. Observe that a directory is not too much different from a domain in structure. The access key method of obtaining access is, however, quite different in spirit from the capability method. Since it is also likely to be more expensive, many systems have a hybrid implementation according to which an object can be accessed once by access key to obtain a capability, which is then used for subsequent accesses. This process when applied to files is usually called *opening* a file [51, Lampson], [52, Lampson].

6.5 Memory Protection

Memory protection hardware is usually closely related to mapping or relocation hardware. There are two aspects to this:

1. Memory which is not in the range of the map cannot be addressed and is therefore protected.
2. In paged or segmented systems (even two-segment ones like the PDP-10) each page or segment in the map may have protection information associated with it.

The point is: each domain must have its own address space, for otherwise there can be no protection [52, Lampson], [53, Lampson]. It is also desirable for one domain to be able to reference the memory of another, subject to the control of the access matrix.

A segmented system in which any segment is potentially accessible from any domain fits well into the framework of Sections 6.2 and 6.3, usually with *A* implemented via capabilities. [24, Dennis and Van Horn], [34, Graham]. It may be an annoyance that a segment capability has a different form than other capabilities, but this problem is minor. Some difficulties may arise, however, with transfers of control from one domain to another, since the addressing hardware will not normally allow the tree addressing and software must be used. [34, Graham].

In the absence of segmentation either pages or files may be treated as objects to be shared. Since the contents of the page map can be changed when changing domains, there is a feasible (though far from elegant) means of sharing memory when necessary while preserving the security of each domain.

In the absence of paging, each domain will have to have private memory which is not accessible to any other domain, except through some ugly circumlocution. The naming problems which result have been considered in Module 5. There is one exception to this observation for the case of nested domains $d_1, \ldots, d_n$ (i.e. $A[d_1\ x] \supset \ldots \supset A\ [d_n, x]$ for all x) on machines with base and bound relocation: simply arrange the memory for the domains contiguously, with d_1 nearest to location 0, and set the bound for d_1 to the total length of all the memory, for d_2 to the total length excluding d_1, etc. Now only a simple addition is required for d_1 to interpret addresses in d_j, $i < j$ [38, Harrison], [53, Lampson].

Module 6: Protection — Topic Outline

6.1 Motivation

- Keep users under control
- Keep programs under control

6.2 Protection Domains

- The idea of different contexts (domains)
- An idealized system to clarify this idea
  - Processes communicating by messages, no sharing
  - Calls and returns are possible
  - Protection is obtained from the system-supplied domain name
- Weaknesses of this system
  - No control over errant processes
  - Need conventions for cooperation between processes

6.3 Objects and Access Matrices—Another Idealized System

- Components of this system—domains, objects, access matrix
- Access attributed
- Changing access
- Relation of protection to the rest of the system

6.4 Implementation Techniques

- Storing the sparse matrix as triples—drawbacks
- Capabilities and C-lists
- Tree-structured naming with capabilities
- Access keys, procedures and lock lists
- Directories
- Hybrid implementations

6.5 Memory Protection

- Memory as an object to be protected
- Segments as objects
- Pages as objects
- Memory protection without mapping
- Nested domains

Module 6: Protection — Reference List Guide

types: C — conceptual, D — descriptive, E — example, T — tutorial

level: S — student, A — advanced student, I — instructor

Key	Author	Type	Level	Importance
14	Crisman (Sec. AD)	E	A	4
22	Dennis	C	A	3
24	Dennis and Van Horn	C	A	1
34	Graham	CE	S	2
37	Hansen	E	S	1
38	Harrison	E	A	3
43	Iliffe	E	A	4
51	Lampson	E	A	4
52	Lampson	CE	S	1
53	Lampson	CE	A	2
55	Linde et al.	E	A	4
62	Molho et al.	E	A	4
78	Vanderbilt	C	I	4
82	Weissman	E	S	2
85	Wilkes (pp 49-59, 75,90)	C	S	1

MODULE 7 — RESOURCE ALLOCATION

7.1 Introduction and Motivation

The previous modules have discussed techniques for process communication, memory management, naming, and protection of objects. Little consideration has been given to the effects of different choices on the attitude of the user and on the performance of the system. Some of the techniques (e.g., multiprogramming) arise from the desire to have a system which is not only correct, but efficient. Every system consists of a set of available resources and a set of users who are constantly demanding to use them. Examples of resources include processor time, memory space, peripherals, channels, and data bases. Examples of consumers at different levels include projects, users, programs, and processes. In a well utilized system, some or all of the resources will be scarce, and systems are considered balanced if all resources are equally scarce. In addition to the requirements of balanced resource usage, efficiency, and smooth operations, most systems seek to allocate resources to maximize some measure of "satisfaction" in the user community. The purpose of studying resource allocation is investigating strategies for allocating resources, and understanding their effects on system efficiency service to the users.

There are two distinct and conflicting goals for resource allocation. 1) Efficiency—measures of equipment utilization, cost of resources, balance, amount of useful computation, throughput, and so on, are to be optimized. 2) Service—measures of user satisfaction, such as turnaround time, consistency and integrity of the system, flexibility, problem solving ability, are to be optimized.

Computer Center managers have tended to stress optimization of throughput in batch-processing systems. Interactive systems are stressing services as well.

Resources can be allocated in two extreme fashions: 1) A user holds all the resources he requires for the entire time he is active, e.g. batch-processing. 2) The system expends at least as many resources optimizing the allocation as the optimization saves. Between (1) and (2) there exists a reasonable compromise. The purpose of an investigation of resource allocation is to approximate the ideal compromise for a particular system.

7.2 Allocation Strategies

In a system environment of scarce resources and demands frequently exceeding the supply, some processes will have to wait. The waiting processes are distributed in a system of queues; a particular process, the scheduler, selects processes from the queues to satisfy objectives of efficiency and service. The scheduler determines how processes flow through the queues and which process obtains a resource when it is free. Schedulers normally use information from three possible sources in making decisions: 1) from the user according to external priorities, 2) from the compiler according to predicted properties of a program, and 3) from the system itself according to its state and the observed behavior of the processes.

It is a policy decision to choose the source of information for the scheduler and how it is used to regulate the progress of processes. Examples of allocation strategies should be presented. There are two areas of particular importance from which they may be drawn.

Processor allocation: Considerable work has been done in allocating processor time to processes, especially in time sharing systems [12, Coffman] and [48, Kleinrock]. Processor allocation disciplines should be discussed and their advantages and disadvantages pointed out qualitatively. Quantitative analysis of the disciplines should be postponed for later on.

Memory Management: Experience shows that memory is one of the most precious resources in modern systems. In Module 4 students were introduced to the concept of memory management policies. If he has not already done so, the instructor should review examples of memory management algorithms, both for nonpaging [50, Knuth, Ch. 2] and for paging environments [18, Denning], [21, Denning]. The properties of the working set model for program behavior should be discussed [18, Denning]. The instructor should point out how most paging systems prepage working sets to a certain extent, viz., upon reinitiation of an interrupted process.

Although these two problems—processor and memory management—have been studied separately in the literature, the instructor should emphasize that there must be a close working relationship between processor and memory management policies, if only because each process demands the use of these resources simultaneously. See [20, Denning] for one view of this relationship, and [83, Wilkes] for another.

7.3 Strategy evaluation

Common sense and intuition are not sufficient to devise a good resource allocation strategy: the interactions among processes and resources are too complex. At the early stages of time sharing, for example, an optimistic approach was commonplace with respect to memory management, virtual memory being considered the answer to the space squeeze problem. The phenomenon of thrashing has made designers aware of the difficulties [19, Denning] and [21, Denning], and has demonstrated the need for careful investigation of the properties of allocation disciplines.

The following is an important trade off between memory utilization and processor utilization, its solution being represented by an optimal degree of multiprogramming. Analysis, rather than intuition, must be used to evaluate it. On the one hand, if many processes are kept active (working sets loaded in memory) the processor has little chance of remaining idle. Since the amount of memory is fixed, each process has less available main storage, and more page faults will be generated. On the other hand, if few processes are kept active there is more than enough space to contain their working sets, but the processor is likely to be excessively idle due to input-output waits blocking the processes.

To evaluate allocation strategies, one uses analysis, experimentation, or both. Analytic models are useful since they typically are computationally convenient to work with, they can be constructed even though the systems being modelled do not exist, and they can be used to predict optimal results. Two classes of models are in use.

Probability Models Probabilistic assumptions are made for the inter-arrival times and magnitudes of demands on a system. The system is analyzed to determine various quantities of interest, such as lengths of queues, waiting times in queues, or efficiency. Probability models have been used most extensively in analyzing queues awaiting service on a processor [60, McKinney], for analyzing the operation of rotating (drum-like) storage devices [1, Abate], [17, Denning], and for analyzing certain aspects of program behavior [21, Denning]. Many of the queueing models for processor scheduling problems have lost their applicability to modern systems: whereas modern systems induce a flow of jobs through a network of queues with many points of congestion (processor, input-output, etc.), the "classical" analyses deal with systems containing only a single point of congestion (processor). No literature on queueing-network models for contemporary systems was available at the time of this writing, but the instructor may wish to investigate the recent literature for such models.

Since many analyses are carried out under the assumption that important probability distributions are exponential, the instructor should review the experimental verification of situations in which exponential assumptions are valid [32, Fuchs].

Deterministic and Discrete Models: Models which do not depend on probability assumptions have been used to analyze resource allocation problems. These include analyses of the deadlock problem [11, Coffman], [35, Habermann], analyses of paging algorithm behavior [58, Mattson], and various analyses of deterministic scheduling problems [57, Manacher].

Since they must be simple to be tractable, analytic models have limitations. There often is a conflict between simplicity on the one hand and realism on the other. Thus, experimental testing and verification is of great importance in dealing with complex situations, though often at considerable expense. There are two kinds of experimental techniques: *simulation* and *a posteriori evaluation*. From a practical point of view, the chief difference between the two techniques is that simulation results can be obtained prior to the implementation of a system; in contrast, a posteriori evaluation allows one to obtain results under actual working conditions. In either case, the effects of various resource allocation strategies can be tested and compared [3, Arden], [56, MacDougall], [73, Saltzer], [87, Wulf].

Analytic models and experimental results are often complementary. Experiments are used to verify assumptions for use in the models. A model can be used to obtain approximate results for checking the reasonableness of simulation results. No experimental work can succeed without some inherent model of system behavior: the presupposed model influences the experimenter in his choice of experiments, level of detail, or choice of parameters.

7.4 Balancing Resources Against Demands

Balanced usage of resources has been found important, since an imbalance (e.g. overload) in the use of one resource can generate an imbalance in the use of others. For example, an overloaded input-output channel can make it impossible to keep the most useful information in main memory. Or, attempted overcommitment of main memory can generate serious under utilization of processor [19, Denning]. Or, a process cannot effectively utilize a processor unless it has a "working-set amount" of memory allocated to it [18, Denning]. These are examples of the general principle that there often exists a critical amount of resource A to make fruitful the allocation of another resource B.

A balanced resource allocation policy implements this principle by regulating membership in the set of active processes such that the total demand of that set matches the available equipment, and the probability of overloading any type of resource is controlled. A wide range of service objectives is implementable within the constraint of a balanced resource allocation policy. Specifying the equipment configuration (relative capacities of the various resource types) is straightforward when a balanced resource allocation policy is used. See [20, Denning].

Module 7: Resource Allocation — Topic Outline

7.1 Introduction and motivation

7.2 Allocation strategies

- Source and nature of priorities
- Processor allocation
- Memory allocation
- Unified approaches to processor-memory allocation

7.3 Strategy Evaluation

- Probability models
- Deterministic and discrete models
- Simulation
- a posteriori evaluation

7.4 Balancing resources against demands

- Critical amount of one resource needed to allocate another
- Balanced resource allocation policy
- Equipment configuration

Module 7:Resource Allocation — Reference List Guide

types: C — conceptual, D — descriptive, E — example, T — tutorial
level: S — student, A — advanced student, I — instructor

Key	Author	Type	Level	Importance
1	Abate and Dubner	C	A	3
3	Arden and Boettner	E	A	3
11	Coffman, et al.	CT	S	1
12	Coffman and Kleinrock	D	S	2
17	Denning	E	I	3
18	Denning	C	A	1
19	Denning	D	S	2
20	Denning	C	A	3
21	Denning	D	S	1
32	Fuchs and Jackson	CE	I	2
35	Habermann	C	A	2
48	Kleinrock	C	I	3
50	Knuth	DE	A	3
56	MacDougall	CD	S	2
57	Manacher	C	A	2
58	Mattson et al	C	A	3
60	McKinney	C	S	1
73	Saltzer and Gintell	C	A	2
83	Wilkes	C	A	3
87	Wulf	E	A	2

This final module treats the aspects of operating systems which are not yet well enough understood to warrant separate treatment. Most of these aspects concern the design, the implementation and the maintenance of systems. Because they are relegated to an inferior position at the end of the course, the instructor should not lead the students to believe that these topics are of less importance. Quite the opposite: pragmatic aspects of system design can make the difference between success and failure. The students, however, can hardly be expected to appreciate what is yet unclear to the expert and unfamiliar to the average lecturer.

As the field of operating systems matures, one would expect these topics to become structured and well understood. It then will be possible for the instructor to treat these topics by means of abstractions, even as the topics of Modules 2-8 have been treated.

8.1 Design

The instructor should review and illustrate several of the current viewpoints on the methodologies of system design, comparing their pros and cons. More than one of the views outlined below may apply to a given system.

The level approach. This methodology is exemplified in the design of the THE system [28, Dijkstra]. One may imagine a series of machines $M_o, M_1, \ldots M_k, \ldots$ and programs $P_1, P_2, \ldots, P_k, \ldots$ such that M_o is the given system hardware and M_k is an extension of M_{k-1} produced by P_k. The extended machines $M_1, M_2, \ldots, M_k, \ldots$ are called the "levels" of the system. Note that a process existing in level k may invoke the services of any process in levels k or lower, but not in any level above k. If the levels are carefully chosen, this approach can have decided advantages with respect to clarity of description and elegance of design. Most important of all, however, this approach lends itself to proving a priori the correctness of the design: one proves by induction that the correctness of M_{k-1} and P_k implies that of M_k. This has clear advantages in the step-by-step construction and debugging of the system. The greatest difficulty with this approach is the problem of choosing what the levels should be.

The top-down approach. The design starts from a general description of the system and, by steadily adding detail, eventually is sufficiently detailed that it can be run on a machine. Conceptually, the design proceeds by successive refinements of system modules, each module being described by its terminal behavior; having proceeded k steps into the design, one proceeds to the $(k+1)^{st}$ step by subdividing some module into a network of simpler modules. Eventually, each module is a simple macro which can be programmed directly on the machine. In practice, this design process consists of successively detailed simulation programs; when the last step is reached the final simulation program is the complete operating system [88, Zurcher]. The advantage of this approach is: the presence of the actual hardware is not required at the beginning; the designer needs only to know its properties. The disadvantage of this approach is a tendency toward infinite regression in the successive refinements of modules, coupled with the possibility that the final set of simple modules may not interface efficiently with the existing hardware. (The task force is not aware of any successful system that has been constructed using this approach alone.)

The nucleus-extension approach. One identifies the minimal elements of an operating system and provides a "nucleus" of programs providing these elements. It is then the responsibility of programmers to add to this—extend it—in ways that meet their requirements. In the RC-4000 system, the nucleus was taken to be the interprocess message transmission facility [36, Hansen], [37, Hansen]. The principal advantage is the ability to get a minimal system operating in a short time; the disadvantage is that more system-development work is placed on the system's users. There is, of course, an analogy between the nucleus of this type system and the lowest levels of a level-structured system.

The modules-interface approach. The system is partitioned as finely as possible into its constituent functions, each function then being implemented as an operating system module. The system designer's task is to identify the modules and specify their interfaces [42, IBM]. This is the method most widely used in systems design. Its chief attraction is that it does not require a great deal of initial planning, as would be required in the three approaches discussed earlier. Experience seems to be quite clear on this point: by allowing the design to be started without adequate preplanning, serious slippage of design deadlines may be the result. (The problems with OS/360, MULTICS, and other large systems are cases in point.) More specifically, designers tend to minimize the complexity of their modules and pay little attention to the complexity of the interfaces. In practice, the resulting interfaces are so complex as to be incomprehensible; worse still, the entire system may be extremely sensitive to change, be unstable, or exhibit bottlenecks in unforeseen places. In other words, it is very hard to predict in advance exactly how the system will behave.

Data base or transaction oriented systems. In certain cases (e.g. reservation or telephone-switching systems) the data base or the transactions to be performed on the system are specified well in advance. The designer has little flexibility, being constrained to build the system to accommodate the existing data base or transaction structures. As an example the Bell Telephone ESS system may be discussed [46, Keister].

Finally there is a collection of ad hoc "seat of the pants" techniques used quite often out of expediency to design and implement systems. They are almost always never successful since they attempt to design a complex system without forethought. The following two can be cited as examples:

The iteration method. Build a version of the system and continuously iterate and modify it to meet the demands and rectify complaints. When this has been done, the result has always been utter confusion for designers, programmers and users.

The deadline method. Get started in an arbitrary fashion, making ad hoc decisions as necessary so that parts of the system are running by specified dates, no matter what. When this has been done, it has normally been necessary to start over again once the deadline is met and the demonstration is given.

8.2 Reliability

Logical correctness and tolerance to error are prime considerations in operating systems design. The reliability requirements placed on many contemporary systems—especially when users become dependent on the system—often mean that the system must be more reliable than the hardware on which it is built. Logical correctness, together with the ability to test and verify correctness, can be invaluable in solving the reliability problem, since then errors can be attributed to hardware failure only. (The THE system is the only system known to the committee which was designed for "correctness" [28, Dijkstra], [29, Dijkstra].)

One important design consideration is that the effect of an error should be localized, so that the fewest users are affected by it and that recovery may be as rapid as possible. The biggest single reliability problem is *integrity*, i.e., protection against loss of information when an error occurs. Four techniques are commonly used to provide integrity. 1) As the importance of data increases, so the probability that an error destroys it should decrease; many designers use the rule of thumb that the fraction of time a given data base is being accessed is inversely proportional to its importance. 2) As the importance of data increases, so should its redundancy. Thus, important system tables can be reconstructed from information in other system tables. The redundant copies should reside in different physical parts of the system. 3) Critical data is checked from time to time for consistency. 4) Incremental dumping is used to copy files into archives, shortly after those files are updated. Thus the archives contain recent copies of all files, should one of the files in the system proper be destroyed. (See [85, Wilkes, pp 86, 90].)

Finally, recovery and restart facilities are an important design consideration and should never be an afterthought. Hardware failure should be considered realistically. The weaknesses of each piece of equipment should be understood clearly. The system should overcome, not enhance, the hardware difficulties [65, NATO, pp 149-153].

8.3 Implementation

Material covering the difficulties encountered during implementation of large software systems can be drawn from the two NATO Reports on Software Engineering [65, NATO], [66, NATO]. Management aspects, and production aspects as well, of large software systems can be emphasized.

Planning. The problems common in the design and implementation of large software packages include personnel, project organization, and human factors [61, Mealy], [65, NATO pp 72-89].

Choice of implementation language. The advantage of a higher level language like PL/1, BCPL, PL/360 cannot be overstated [65, NATO pp 55-59].

Portability. The production of software as machine independent as possible will eventually enable programs to be transported from one installation to another. [66, NATO pp 28-33].

Debugging. Debugging facilities and check-out procedures are invaluable to programmers and implementors. [66, NATO pp 23-25].

Documentation. Good documentation on production and service is extremely important [65, NATO pp 116-117, pp 209-211]. Automatic flowcharting techniques are often used to assist in documenting programs [2, Abrams].

8.4 Performance Monitoring and Evaluation

The field of measurement, evaluation, and tuning of operating systems is progressing rapidly [9, Calingaert], [39, Hart], [65, NATO pp 200-203]. The relative advantages of hardware versus software measuring tools should be discussed, [79, Warner], but the problem of data reduction is beyond the scope of this course. Techniques for regulating progress of processes and for "tuning" the system are related both to performance monitoring and to resource allocation [88, Wulf], [20, Denning].

Again we stress that the students should realize that these problems are very important. That there are no unifying abstractions to be presented does not necessarily mean that the topics should be ignored. The students should be aware of the difficulties and the existence of limited solutions, even ad hoc ones.

Module 8: Pragmatic Aspects – Topic Outline

8.1 Design

- Levels approach
- Top-down approach
- Nucleus-extension approach
- Modules-interface approach
- Data base or transaction oriented
- "Seat of the pants" techniques

8.2 Reliability

- Design for reliability
- System Integrity

8.3 Implementation

- Personnel and project organization
- Implementation languages
- Portability
- Debugging
- Documentation

8.4 Performance Measurement and Evaluation

- Measurement
- Evaluation and Tuning

Module 8: Pragmatic Aspects – Reference List Guide

types: C – conceptual, D – descriptive, E – example, T – tutorial

level: S – student, A – advanced student, I – instructor

Key	Author	Type	Level	Importance
2	Abrams	D	S	4
9	Calingaert	D	A	3
20	Denning	C	I	3
28	Dijkstra	D	S	1
29	Dijkstra	C	A	1
36	Hansen	D	A	2
37	Hansen	D	S	1
39	Hart	D	S	3
42	IBM	D	I	3
46	Keister	D	S	2
61	Mealy	D	A	3
65	NATO	D	S	2
66	NATO	D	S	2
79	Warner	D	S	3
85	Wilkes	T	S	1
87	Wulf	D	A	3
88	Zurcher and Randell	C	A	2

BIBLIOGRAPHY

The following abbreviations are used in the bibliography.

ACM – Association for Computing Machinery
IEEE – Institute for electrical and electronics engineers
IEEETC – IEEE Transactions on Computers
CACM – Communications of the ACM
JACM – Journal of the ACM
CS – Computing Surveys (ACM)
FJCC – Fall Joint Computer Conference
SJCC – Spring Joint Computer Conference
2SOSP – Second Symposium on Operating Systems Principles (proceedings available from ACM, 1133 Avenue of Americas, New York, N.Y. 10036)

1. Abate, J., and Dubner, H. Optimizing the Performance of a Drum-Like Storage. *IEEE Trans. C-18*, 11 (Nov. (Nov. 1969), 992-997.
2. Abrams, M. A Comparative Sampling of the Systems for Producing Computer-Drawn Flowcharts. *Proc. 1968 ACM Nat. Conference.*
3. Arden, B. and Boettner, D. Measurement and Performance of a Multiprogrammed System. Proc. 2SOSP (Oct. 1969).
4. Belady, L.A. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Sys. J. 5*, 2 (1966), 78-101.
5. Bell, C.G. and Newell, A. *Computer Structures: Readings and Examples*. Chapt. 3, McGraw Hill (1971).
6. Bensoussan, A., Clingen, C.T., and Daley, R.C. The Multics Virtual Memory. Proc. 2SOSP (Oct. 1969).
7. Bernstein, A.J. et al. Process Control and Communication. Proc. 2SOSP (Oct. 1969).
8. Bétourné, C., et al. Process Management and Resource Sharing in the Multiaccess System 'ESOPE'. *CACM 13*, 12, (Dec. 1970).
9. Calingaert P. System Performance Evaluation Survey and Appraisal. *CACM* (Jan. 1967).
10. Clark, W.A. The Functional Structure of OS/360, Part III: Data Management. *IBM Sys. J. 5*, 1 (1966), 30-51.
11. Coffman, E.G., Elphick, M., and Shoshani, A. System Deadlocks. *CS 3*, 2 (June 1971).
12. Coffman, E., and Kleinrock L. Computer Scheduling Measures and Their Countermeasures. *AFIPS Conf. Proc. 32 (1968 SJCC).*
13. Corbató, F.J. and Vyssotsky, V.A. Introduction and Overview of the Multics System. *AFIPS Conf. Proc. 27* 1965 FJCC), 185-196.
14. Crisman, P.A. *The Compatible Time-sharing System; A Programmer's Guide.* MIT Press, 2nd Edition, Cambridge, Mass. (1965).
15. Daley, R.C., and Dennis, J.B. Virtual Memory, Processes, and Sharing in MULTICS. *CACM 11*, 5 (May 1968), 306-312.
16. Daley, R.C., and Neumann, P.G. A General-Purpose File System for Secondary Storage. *AFIPS Conf. Proc. 27* (1965 FJCC), 213-229.
17. Denning, P.J. Effects of Scheduling on File Memory Operations. *AFIPS Conf. Proc. 30* (1967 SJCC), 9-22.
18. Denning, P.J. The Working Set Model for Program Behavior. *Comm. ACM 11*, 5 (May 1968), 323-333.
19. Denning, P.J. Thrashing: Its Causes and Prevention. *AFIPS Conf. Proc. 33* (1968 FJCC), 915-922.
20. Denning, P.J. Equipment Configuration in Balanced Computer Systems. *IEEETC C-18* (Nov. 1969), 1008-1012.
21. Denning, P.J. Virtual Memory. *CS 2*, 3 (Sept. 1970), 153-189.
22. Dennis, J.B. Segmentation and the Design of Multiprogrammed Computer Systems. *JACM 12*, 4 (Oct. 1965), 589-602.
23. Dennis. J.B. A Position Paper on Computing and Communications. *CACM 11*, 5 (May 1968), 370-377.
24. Dennis, J.B., and Van Horn, E.C. Programming Semantics for Multiprogrammed Computations. *CACM 9*, 3 (Mar. 1966), 143-155.
25. Dennis, J.B. et. al. *Computation Structures*. Chapt. 8, Class notes for Subject 6.232, MIT (1970). [Available from: MIT, Rm. 4-213, Cambridge, Mass. 02139.]
26. Dijkstra, E.W. Solution of a Problem in Concurrent Programming Control. *CACM 8*, 9 (Sept 1965), 569.
27. Dijkstra, E.W. Cooperating Sequential Processes. *Programming Languages* (F. Genuys, ed.), Academic Press (1968), 43-112.
28. Dijkstra, E.W. The Structure of THE Multiprogramming System. *CACM 11*, 5 (May 1968), 341-346.
29. Dijkstra E. A constructive approach to the problem of program correctness. *BIT 8* (1968).
30. Fano, R.M. and Corbató, F.J. Time Sharing on Computers *Sci. Amer. 215*, 3 (Sept. 1966), 129-140.
31. Foster, J.M. *List processing*. Macdonald and Co., London (1967).
32. Fuchs, E., and Jackson, P.E. Estimates of Random Variables for Certain Computer Communications Traffic Models. *CACM 13*, 12, (Dec. 1970).
33. Gear, C.W. *Computer Organization and Programming*. McGraw Hill (1969).
34. Graham, R.M. Protection in an information Processing Utility. *CACM 11*, 5 (May 1968),368.
35. Habermann, N. Prevention of System Deadlocks. *CACM 12,* (July 1969).

36. Hansen, P.B. (ed.) *RC4000 Software Multiprogramming System*, A/S Regnecentralen, April 1969, Falkoner Alle 1, Copenhagen F. Denmark.

37. Hansen, P.B. The Nucleus of a Multiprogramming System, *CACM 13* (April 1970), 238.

38. Harrison, M.C. Implementation of the SHARER2 Time-Sharing System. *CACM 11*, 12 (Dec. 1968), 845.

39. Hart, L.E. The User's Guide to Evaluation Products, *Datamation 17*, 1 (Dec. 1970).

40. Hellerman, H. *Digital Computer System Principles.* McGraw Hill (1967).

41. Horning J.J., and Randell, B. Structuring Complex Processes, IBM Research Rpt. RC2459 (May 2, 1969), IBM Research Center, P.O. Box 218, Yorktown Heights, N.Y. 10598.

42. IBM. OS/360 Concepts and Faciiities. In *Programming Languages and Systems*. (S. Rosen, Ed.), McGraw-Hill (1967), 598.

43. Iliffe, J.K. *Basic Machine Principles*. American Elsevier (1968).

44. Iliffe, J.K., and Jodeit, J.G. A Dynamic Storage Allocation Scheme. *The Computer Journal* (Oct. 1962), 200-209.

45. Jodeit, J.G. Storage Organization in Programming Systems, *CACM 11*, 11 (Nov. 1968), 741-746.

46. Keister, W. et al. No. 1 ESS: System Organization and Objectives. *Bell System Technical Journal* 43, 5 (Sept. 1964), 1831-1844.

47. Kilburn, T. et al. One-Level Storage System. *IRE Trans. EC-11*, 2 (Apr. 1962), 223-235.

48. Kleinrock, L. A Continuum of Time Sharing Algorithms. *Proc. AFIPS Conf. 36* (1970 SJCC)

49. Knuth, D.E. Additional Comments on a Problem in Concurrent Programming. *CACM 9* (May 1966), 321-323.

50. Knuth, D.E. *The Art of Computer Programming* (Vol. 1). Addison- Wesley (1968), Ch. 2.

51. Lampson, B.W., et al. A User Machine in a Time-sharing System. *Proc. IEEE 54*, 12 (Dec. 1966).

52. Lampson, B.W. Dynamic Protection Structures. *Proc. AFIPS Conf. 35* (1969 FJCC).

53. Lampson, B.W. On Reliable and Extensible Operating Systems. *Infotech State of the Art Proceedings* (1970).

54. Lanzano, B.C. Loader Standardization for Overlay Programs. *CACM 12*, (Oct. 1969), 541-550.

55. Linde, R.R., et al. The ADEPT-50 Time Sharing System. *Proc. AFIPS Conf. 35* (1969 FJCC).

56. MacDougall, M.H. Computer System Simulation: An Introduction. *CS 2*, 3 (Sept. 1970), 191-210.

57. Manacher, G.K. Production and Stabilization of Real-Time Task Schedules. *JACM 14*, 3 (July 1967) 439-465.

58. Mattson, R.L., Gecsei, J., Slutz, D.R., and Traiger, I.L. Evaluation Techniques for Storage Hierarchies. *IBM Sys. J. 9*, 2 (1970), 78-117.

59. McCarthy, J., Corbato, F.J., and Daggett, M.M. The Linking Segment Subprogram Language and Linking Loader. *CACM 6*, 7 (July 1963), 391-395. In Rosen, S. (ed), *Programming Languages and Systems*, McGraw Hill (1967).

60. McKinney, J.M. A Survey of Analytical Time-Sharing Models. *CS 1*, 2 (June 1969) 105-116.

61. Mealy, G. The System Design Cycle. *Proc. 2SOSP.* (Oct. 1969).

62. Molho, L. Hardware Aspects of Secure Computing. *Proc. AFIPS Conf. 36* (1970 SJCC), 135.

63. Morris, R. Scatter Storage Techniques. *CACM 11*, 1 (Jan. 1968), 38-44.

64. Murphy, J.E. Resource Allocation with Interlock Detection in a Multi-task System. *AFIPS Conf. Proc. 33* (1968 FJCC), 1169-1176.

65. NATO Report on SOFTWARE ENGINEERING, Garmish, Oct. 1968. Available free of charge through NATO, Dr. H. Arnth-Jensen, Scientific Affairs Division, OTAN/NATO, 1110 Bruxelles, Belgium.

66. NATO Report on SOFTWARE ENGINEERING, Rome, Oct. 1969. Available free of charge through NATO, Dr. H. Arnth-Jensen, Scientific Affairs Division, OTAN/NATO, 1110 Bruxelles, Belgium.

67. Naur, P., et al. Revised ALGOL Report. *CACM 6*, 1 (Jan. 1963), 1-17. In Rosen, S. (ed.), *Programming Languages and Systems,* McGraw (1967).

68. Pankhurst, R.J. Program Overlay Techniques. *CACM 11*, 2 (Feb. 1968), 119-125.

69. Parkhill, D. *Challenge of the Computer Utility.* Addison Wesley (1966).

70. Randell, B., and Kuehner, C.J. Dynamic Storage Allocation Systems. *Comm. ACM 11*, 5 (May 1968), 297-305.

71. Randell, B. and Russell, L.J. *ALGOL 60 Implementation*. Academic Press (1964).

72. Saltzer, J.H. *Traffic Control in a Multiplexed Computer System.* MIT Project MAC Rpt MAC-TR-30 (1966), Project MAC, 545 Technology Square, Cambridge, Mass. 02139.

73. Saltzer, J., and Gintell, J. The Instrumentation of MULTICS. *CACM 13*, (Aug. 1970).

74. Sayre, D. Is Automatic Folding of Programs Efficient Enough to Displace Manual? *CACM 12*, 12 (Dec. 1969), 656-660.

75. Sharpe, W.F. *The Economics of Computers*. (Chapt. 10). Columbia University Press (1969).

76. Spier, M.J., and Organick, E.I. The MULTICS Inter-process Communication Facility. *Proc. 2SOSP.* (Oct. 1969).

77. Subcommittee of the American Standards Association Sectional Committee X3, Computers and Information Processing. FORTRAN vs. Basic FORTRAN—A Programming Language for Information Processing on Automatic Data Processing Systems. *CACM 7*, 10 (Oct. 1964), 591-624.

78. Vanderbilt, D.H. Controlled Information Sharing in a Computer Utility. MAC-TR-67, MIT, October 1969, Document Room, Project MAC, 545 Technology Square, Cambridge, Mass. 02139.

79. Warner, C.D. Monitoring: A Key to Cost Efficiency. *Datamation 16*, 17 (Jan. 1971).

80. Wegner, P. Communication Between Independently Translated Blocks. *CACM* (July 1962), 376-381.

81. Wegner, P. *Programming Languages, Information Structures and Machine Organization*. (Sections 4.1-4.7), McGraw Hill (1968).

82. Weissman, C. Security Controls in the ADEPT-50 Time-Sharing System. *Proc. AFIPS Conf. 35* (1969 FJCC), 119.

83. Wilkes, M.V. A Model for Core Space Allocation in a Time Sharing System. *AFIPS Conf. Proc. 34* (1969 SJCC), 265-271.

84. Wilkes, M.V. Slave Memories and Dynamic Storage Allocation. *IEEE Trans. EC-14* (Apr. 1965), 270-271.

85. Wilkes, M.V. *Time-Sharing Computer Systems*. Am. Elsevier (1968).

86. Wirth, N. On Multiprogramming Machine Coding and Computer Organizations. *CACM 12*, 9 (Sept. 1969), 489-498.

87. Wulf, W.A. Performance Monitors for Multiprogramming Systems. *Proc. 2SOSP* (Oct. 1969).

88. Zurcher, F. and Randell, B. Iterative Multi-Level Modelling. A Methodology for Computer System. *IFIP Congress 68*, Edinburgh, Scotland, (Aug. 1968).