Third Generation Computer Systems

PETER J. DENNING

Princeton University, Princeton, New Jersey 08540*

The common features of third generation operating systems are surveyed from a general view, with emphasis on the common abstractions that constitute at least the basis for a "theory" of operating systems. Properties of specific systems are not discussed except where examples are useful. The technical aspects of issues and concepts are stressed, the nontechnical aspects mentioned only briefly. A perfunctory knowledge of third generation systems is presumed.

Key words and phrases: multiprogramming systems, operating systems, supervisory systems, time-sharing systems, programming, storage allocation, memory allocation, processes, concurrency, parallelism, resource allocation, protection

CR categories: 1.3, 4.0, 4.30, 6.20

NTRODUCTION

It has been the custom to divide the era of electronic computing into "generations" whose approximate dates are:

First: 1940–1950;
Second: 1950–1964;
Third: 1964-present; and
Late Third: 1968-present.

The principal properties of the generations are summarized in Table I. The term "generation" came into wide use after 1964, the year in which third generation machines were announced. Although the term was originally used to suggest differences in hardware technology, it has come to be applied to the entire hardware/software system rather than to the hardware alone [R3, R4].

A Definition of "Operating System"

As will be discussed in detail below, the term "process" is used to denote a program in execution. A computer system may be defined in terms of the various supervisory and control functions it provides for the processes created by its users: 1) creating and removing processes; 2) controlling the progress of processes—i.e., ensuring that each logically enabled process makes progress at a positive rate and that no process can indefinitely block the progress of others; 3) acting on exceptional conditions arising during the execution of a process—e.g., arithmetic or machine errors, interrupts, addressing snags, illegal and privileged instructions, or protection violations; 4) allocating hardware resources among processes; 5) providing access to software resources—e.g., files, editors, compilers, assemblers, subroutine libraries, and programming systems; 6) providing protection, access control, and security for information; and 7) providing interprocess communications where required. These functions must be provided by the system because they cannot be handled adequately by the processes themselves. The computer system software that assists the hardware in implementing these functions is known as the *operating system.*

Two points about operating systems should be noted. First, users seldom (if ever) perform a computation without assistance from the operating system; thus, they often

* Department of Electrical Engineering. Work reported herein was supported in part by NASA Grant NGR-31-001-170 and by NSF Grant GY-6586.

CONTENTS

come to regard the entire hardware/software system, rather than the hardware alone, as "the machine." Secondly, in extensible systems, such as MULTICS [A8, C5, S1] or the RC-4000 [B4, B5], a user may redefine or add to all but a small nucleus of operating system programs; thus, an operating system need not be fixed or immutable, and each user may be presented with a different "machine."

Types of Systems

An enormous variety of systems are regarded as members of the third generation. The range includes general-purpose programming systems, real-time control systems, time-sharing systems, information service and teleprocessing systems, and computer networks; and noninteractive, large batch-processing systems, such as the Chippewa Operating System on the CDC 6600 or OS on the IBM 360 series. It also includes a wide range of interactive systems, of which there are five increasingly sophisticated categories [D10]: 1) *dedicated information systems*, such as airline and other ticket reservation systems, in which the users may perform a limited number of "transactions" on a given data base; 2) *dedicated interactive systems*, such as Joss (Johnniac Open Shop System) or QUIKTRAN, in which the user may program transactions within a given language; 3) *general-purpose interactive systems*, in which users may write programs in any one of a given set of languages (most time-sharing service bureaus offer systems of this type); 4) *extensible systems*, such as MIT's CTSS (Compatible Time Sharing System) [C4, C8], in which users are not restricted to the programming languages and programming systems provided by the system, but may develop their own and make them available to other users; and 5) *coherent systems*, such as MIT's MULTICS (MULTiplexed Information and Computing Service) [C5], in which one may construct new programs or programming systems from modules by various authors in various languages, without having to know the internal operation of any module.

The views programmers and designers

TABLE I. A SUMMARY OF CHARACTERISTICS OF THE GENERATIONS OF COMPUTERS

Characteristic	*Generations*			
	First	*Second*	*Third*	*Late third*
Electronics:				
Components	vacuum tubes	transistors	integrated circuits	same as third
Time/operation	0.1–1.0 msec	1–10 μsec	0.1–1.0 μsec	same as third
Main memory:				
Components	electrostatic tubes and delay lines	magnetic drum and magnetic core	magnetic core and other magnetic media	semiconductor registers (cache)
Time/access	1 msec	1–10 μsec	0.1–10 μsec	0.1 μsec
Auxiliary memory	paper tape, cards, delay lines	magnetic tape, disks, drums, paper cards	same as second, plus extended core and mass core	same as third
Programming languages and capabilities	binary code and symbolic code	high-level languages, subroutines, recursion	same as second, plus data structures	same as third, plus extensible languages and concurrent programming
Ability of user to participate in debugging and running his program	yes (hands-on)	no	yes (interactive and conversational programs)	same as third
Hardware services and primitives	arithmetic units	floating-point arithmetic, interrupt facilities, microprogramming, special-purpose I/O equipment	same as second, plus: microprogramming and read-only storage, paging and relocation hardware, generalized interrupt systems, increased use of parallelism, instruction lookahead and pipelining, datatype control	
Software and other services	none	subroutine libraries, batch monitors, special-purpose I/O facilities	same as second, plus: multiaccessing and multiprogramming, time-sharing and remote access, central file systems, automatic resource allocation, relocation and linking, one-level store and virtual memory, segmentation and paging, context editors, programming systems, sharing and protection of information	

take of systems are almost as varied as the systems themselves. Some are viewed as large, powerful batch-processing facilities offering a wide variety of languages, programming systems, and services (e.g., IBM's OS/360 and CDC's Chippewa/6600). Some are viewed as efficient environments for certain programming languages (e.g., ALGOL on the Burroughs B6500). Some are viewed as extensions of some language or machine (e.g., the virtual machines presented to users in IBM's CP/67 or M44/44X). Others are viewed as information management systems (e.g., SABRE Airline Reservations System). Still others are viewed as extensible systems or information utilities (e.g., MIT's CTSS and MULTICS, A/S Regnecentralen's RC-4000, IBM's 360/TSS, and RCA's Spectra 70/46).

Common Properties of Systems

Despite the diversity of system types and views about them, these systems have a great deal in common. They exhibit common characteristics because they were designed with common general objectives, especially to provide programmers with: 1) an efficient environment for program development, debugging, and execution; 2) a wide range of problem-solving facilities; and 3) low-cost computing through the sharing of resources and information. The characteristics we shall describe below are properties of the class of third generation systems,

but no particular system need exhibit all of them.

Concurrency

A first common characteristic of third generation systems is concurrency—i.e., the existence or potential existence of several simultaneous (parallel) activities or *processes*. The term "process" was introduced in the early 1960s as an abstraction of a processor's activity, so that the concept of "program in execution" could be meaningful at any instant in time, regardless of whether or not a processor was actually executing instructions from a specific program at that instant. The term "process" is now used in a more general sense to denote any computation activity existing in a computer system. Thus, the idea of *parallel processes* can be interpreted to mean that more than one program can be observed between their starting and finishing points at any given time. Processes may or may not be progressing simultaneously; at a given time, for example, we may observe one process running on the central processor, a second suspended awaiting its turn, and a third running on an input/output channel. Processes interact in two ways: indirectly, by competing for the same resources; and directly, by sharing information. When processes are logically independent they interact indirectly, and control of concurrency is normally delegated to the underlying system; but when they are interacting directly, control of concurrency must be expressed explicitly in their implementations.

Since an operating system is often regarded as a control program that regulates and coordinates various concurrent activities, the need for ways of describing concurrent activity at a programming-language level is felt most acutely by systems programmers. There are at least three reasons for this: 1) the demand for rapid response time and efficient equipment utilization has led to various forms of resource sharing, and has created a need for dynamic specification of the resource requirements of a program; 2) widespread use of concurrent activity between the central machine and its peripheral devices already exists; and 3) the desire to share information and to communicate among executing programs has led to the development of message transmission facilities and software mechanisms for stopping a program while it awaits a signal from another.

Automatic Resource Allocation

A second common characteristic of third generation systems is the existence of an automatic resource allocation mechanism with a wide variety of resources. The reasons for central system control of resource allocation include the following. 1) Programmers tend to be more productive when they do not have to be concerned with resource allocation problems in addition to the logical structures of their algorithms. Moreover, programming languages shield programmers from details of machine operation. 2) Due to the unpredictable nature of demands in a system supporting concurrent activity, a programmer is at a disadvantage in making efficient allocation decisions on his own. 3) A centralized resource allocator is able to monitor the entire system and control resource usage to satisfy objectives both of good service and system efficiency.

Several systems have incorporated a very general view of the nature of resources themselves. As we shall see, this view allows one to treat many aspects of system operation as resource allocation problems, and with good results. According to this view, the set of objects that a process may use, and on which its progress depends, is called the *resources* required by the process. The system provides a variety of resource *types*, and there is generally a limited number of *units* of each type available. Examples of resource types include: processors, memories, peripheral devices, data files, procedures, and messages. Examples of resource units include: a processor, a page of memory, a disk or drum track, a file, a procedure, or a message. Some resource types have immediate realizations in terms of hardware (e.g., processor, memory, peripherals), whereas others are realized in software only (e.g., files, procedures, messages). Some resource types are "reusable," a unit of that type being reassignable after its release; whereas other

types are "consumable," meaning they cease to exist after use (e.g., a message or the activation record of a called procedure). All resources have internal "states" that convey information to their users; if the state of a resource is modified during use, a unit of that type is said to be "subject to exclusive control" by, at most, one process at a time. A unit resource is "preemptible" only if the system may intervene and release it without losing input, output, or progress of the process involved. A non-preemptible resource may be released only by the process to which it is assigned.

Sharing

A third common characteristic of third generation systems is sharing, the simultaneous use of resources by more than one process. The term "sharing" has two related meanings in today's parlance. First, it denotes the fact that resource types can be shared regardless of whether or not individual units of that type can be shared. In fact, most units of physical resource types are subject to exclusive control by the processes to which they are assigned, and some form of multiplexing must be used to implement the sharing, but more will be said about this shortly. Secondly, the term "sharing" refers to the sharing of information, potentially or actually. The desire to share information is motivated by three objectives:

1) *Building on the work of others:* the ability to use, in one's own programs, subprograms or program modules constructed by others.
2) *Shared-data problems:* the sharing by many users of a central data base, which is required by certain types of problems (e.g., ticket reservations).
3) *Removing redundancy:* as many system software resources (compilers or input/output routines, for example) may be required simultaneously by several active processes, and as giving each process its own copy would tend to clutter the already scarce memory with many copies of the same thing, it is desirable that the system provide one shared copy of the routine for all to use.

Achieving these objectives places new requirements on the system and its compilers. Point 1 requires that compilers provide linkage information with compiled modules, and that there be loaders to establish intermodular linkages at the proper time. Point 2 requires some sort of "lockout" mechanism* so that no process could attempt to write or read a part of the data base that is being modified by another process. A program module may satisfy point 3 by being "serially reusable": i.e., the compiler would have to have inserted a "prologue" into it which would initialize the values of any internal variables that might have been changed by a previous use of the module; or a lockout mechanism may be required to prevent a second process from using the module before a first has finished.

To obtain sharable (not merely reusable) subprograms, the compiled code must be partitioned into "procedure" and "data." The procedure part is read-only, and contains all the instructions of the program. The data part contains all the variables and operands of the program and may be read or written. Each process using such code may be linked to a single, common copy of the procedure part, but must be provided with its own private copy of the data part. The techniques for compiling code in this manner were originally devised for the ALGOL programming language to implement recursive procedures (i.e., procedures that can call themselves). Since a recursive subroutine can be called (activated) many times before it executes a single return, and since each call must be provided with parameter and internal variable values different from other calls, it is necessary to associate private working storage (an activation record) with each instance of an activated procedure. (More will be said about this in the next section.) At any given time, one may inspect an ALGOL program in execution and find many instances of procedure activations; thus, there is an immediate analogy between this notion and the notion of parallel processes discussed above. For this reason, many of

* Details of lockout mechanisms will be discussed in connection with the mutual exclusion problem in the fourth section, "Concurrent Processes."

the techniques for handling multiple activations of a procedure in a single program (recursion) were found to be immediately applicable to multiple activations of a procedure among distinct programs (sharing). Because the names and memory requirements of all procedures that might be used by a process may not be known in advance of execution (as is the case with the procedures in ALGOL programs), the mechanism for linking procedures and activation records among processes can be more complicated and may require special hardware to be efficient [D1].

Multiplexing

A fourth common characteristic of third generation systems is multiplexing, a technique in which time is divided into disjoint intervals, and a unit of resource is assigned to, at most, one process during each interval [S1]. As mentioned above, multiplexing is necessitated by the desire to share a given resource type when its individual units must be controlled exclusively by processes to which they are assigned. Multiplexing has assumed particular importance as a means, not only of maintaining high load factors on resources, but also of reducing resource-usage costs by distributing them among many users. The time intervals between the instants of reassignment of the multiplexed resource may be defined naturally (by the alternation of a process between periods of demand and nondemand) or artificially (by means of "time slicing" and preemption). The latter method is used primarily in time-sharing and other systems in which response-time deadlines must be satisfied. Although multiplexing is not sharing, it can be used to give the appearance of sharing; if, for example, a processor is switched equally and cyclically among n programs with sufficient speed, a human observer could not distinguish that system from one in which each program has its own processor of $1/n$ speed.

Let us digress briefly to describe some specific examples of commonly used techniques for sharing and multiplexing.

1) Under *multiprogramming*, (parts of) several programs are placed in main memory at once; this not only makes better use of main memory, but it maintains a supply of executable programs to which the processor may switch should the program it is processing become stopped. Moreover, a particular user's program need not reside continuously in memory during the entire length of its run time; indeed, it may be swapped repeatedly for some other program requiring use of the memory—one that is residing in auxiliary memory waiting its turn. Swapping takes place without any overt action by the programs in memory; to the user its only observable effect is that the machine may appear to be operating more slowly than if he had it entirely to himself. Implementing multiprogramming requires two fundamental alterations in system organization from second generation systems: (a) since programs cannot be allowed to reference regions of memory other than those to which they have been granted access, and since the exact region of memory in which a program may be loaded cannot be predicted in advance, programming must be location-independent; and (b) special mechanisms are required to preempt the processor from one program and switch it to another without interfering with the correct operation of any program.
2) Under *multiaccessing*, many users are permitted to access the system (or parts of it) simultaneously. A simple multiaccess system is the batch-processing system with several remote-job-entry terminals. An advanced multiaccess system is the time-sharing system ("time multiplexing" would be more accurate), in which many users at many consoles use system resources and services conversationally. It should be noted that multiaccessing and multiprogramming are independent concepts; many CDC 6600 batch-processing installations are examples of multiprogrammed systems with one job-entry terminal, whereas MIT's CTSS [C4] is an example of a monoprogrammed time-sharing system that uses swapping to implement memory sharing.
3) *Multitasking* refers to the capability of a system to support more than one active process. Multitasking must necessarily

exist in multiprogramming systems, but it may or may not exist in multiaccess systems. It may also exist in batch-processing systems under the auspices of certain programming languages, such as PL/I.

4) Under *multiprocessing*, the system supports several processors so that several active processes may be in execution concurrently. If one regards channels and other input/output controllers as special-purpose processors, then every third generation system (and many second generation systems as well) is a multiprocessor system. However, the term "multiprocessor" is normally applied only to systems having more than one central processor; examples of such systems are the ILLIAC IV, MULTICS, the Bell System's ESS (Electronic Switching System), and the Michigan Time Sharing System on the IBM 360/67.

Remote Conversational Access

A fifth common characteristic of third generation systems is remote conversational access, in which many users are allowed to interact with their processes. Interactive, or on-line, computing is now known to make rapid program development and debugging possible [D10, W8]. It is required for shared-data-base problems—e.g., ticket reservation systems.

Nondeterminacy

A sixth common characteristic of third generation systems is nondeterminacy, i.e., the unpredictable nature of the order in which events will occur. Nondeterminacy is a necessary consequence of concurrency, sharing, and multiplexing. The order in which resources are assigned, released, accessed, or shared by processes is unpredictable, and the mechanisms for handling concurrency, sharing, and multiplexing must be designed to allow for this. In contrast to global, system-wide nondeterminacy, there is often a need for local forms of determinacy at the programming level. When a collection of processes is cooperating toward a common goal and sharing information, it is usually desirable that the result of their computation depend only on the initial values of the data, not on their relative speeds. A system of processes having this property is sometimes called "determinate" or "speed-independent." This problem is considered again in the fourth section, "Concurrent Processes."

Long-Term Storage

A seventh common property of third generation systems is the presence of long-term storage. Second generation systems provided it in the limited form of subroutine libraries (no provision was made for data files). Since these libraries were limited in size and were accessible only via compilers or loaders, the problems of managing them were not severe. However, many third generation systems endeavor to provide users with means of storing their own information in the system for indefinite periods. This gives rise to three new, nontrivial problems: 1) there must be a file system to manage files of information entrusted to the system; 2) the system must provide reasonable guarantees for the survival of a user's information in case of system failure or even the user's own mistakes; and 3) the system must control access to information in order to prevent unauthorized reading or writing.

Abstractions Common to Systems

The common properties discussed above have given rise to abstractions about third generation system organization; by "abstraction" we mean a general concept or principle describing a problem area, from which most implementations can be deduced. The abstractions can be regarded as forming at least a basis for a "theory" of operating systems principles.

The five areas in which the most viable abstractions have evolved are

1) programming;
2) storage allocation;
3) concurrent processes;
4) resource allocation; and
5) protection.

The next five sections of this paper study these abstractions and some of their consequences in detail. Other areas, about which

few viable abstractions have evolved, will be discussed in the last section of the paper.

PROGRAMMING

It is generally agreed that the capabilities of the hardware in computer systems should be extended to allow efficient implementation of certain desirable programming language features. For this reason, programming objectives have had a profound influence on the development of computer and programming systems. The four most important of these are discussed in the following paragraphs.

1) *High-level languages:* The introduction of FORTRAN in 1956 and ALGOL in 1958 marked the beginning of a period of intense activity in high-level language development. From the first it was recognized that a programmer's problem-solving ability could be multiplied by high-level languages, not only because such languages enable him to express relatively complicated structures in his problem solution as single statements (contrasted with the many instructions required in machine language), but because they free him from much concern with machine details. The enormous variety of languages testifies to the success of this effort: algebraic languages, block-structured languages, procedure-oriented languages, string-manipulation languages, business-oriented languages, simulation languages, extensible languages, and so on [S2]. During the late 1950s there raged often bitter debates over the efficiency of compilation and execution of high-level language programs as compared to machine-language programs [W5]. These arguments subsided during the 1960s. Now it is generally agreed that compiler-generated code (and the compiling processes themselves) can be made of acceptable quality, and that any efficiency lost by the use of such code is handsomely compensated for by increased programmer productivity.
2) *Program modularity:* This term refers to the ability to construct programs from independently preparable, compilable, testable, and documentable "modules" (subprograms) that are not linked together into a complete program until execution. The author of one module should be able to proceed with its construction without requiring knowledge of the internal structure, operation, or resource requirements of any other module.
3) *Machine independence:* This term refers to the ability of a programmer to write programs without having to worry about the specific resource limitations of a system. It has been used according to one or more of three interpretations: (a) *processor independence*—appropriate compiling techniques enable a single program text to be prepared for execution on a processor with an arbitrary instruction set; (b) *memory independence*—a programmer may write a subprogram without having to know how much main memory might be available at execution time or what the memory requirements of (perhaps yet unwritten) companion subprograms might be; and (c) *I/O independence*—programmers obtain input (create output) by reading (writing) "I/O-streams," a given stream being attachable through interface routines to any device at execution time. A significant degree of machine independence of these three types is already being afforded by high-level languages, yet the techniques are only just beginning to be far enough advanced to permit program "transportability"—i.e., the ability to move a program or programming system written in any language from one machine to another [P2, W1].
4) *Structured data capabilities:* Programming languages capable of handling problems involving structured data (e.g., linked lists, trees) have been increasingly important, especially since the mid 1960s.

These four programming objectives are achieved to varying degrees in most modern computer and programming systems. Achieving them places new responsibilities on the operating system and on the compilers. They enable the creation of programs whose memory requirements cannot be predicted prior to loading time, necessitating

some form of dynamic storage allocation. To achieve a wide variety of languages, there must be a corresponding variety of compilers and a file system for storing and retrieving them. To achieve program modularity, the system and its compilers must provide a "linking" mechanism for connecting modules and establishing communications among them [M3]. This mechanism is normally provided by a "linker" or "linking loader" and can become quite complicated, especially when sharing is permitted [D1, S1]. To achieve memory independence, the system must provide a mechanism for translating program-generated memory references to the correct machine addresses, this correspondence being indeterminable a priori. This mechanism is normally provided in the form of a "relocating loader," "relocation hardware," or "virtual memory." To achieve structured data capability, the system must be able to allocate new storage on demand (should data structures expand) and deallocate storage as it falls out of use (should data structures contract).

Systems Programming Languages

There is an interesting parallel between events prior to 1965 and those following. Just as the question of machine-languages versus high-level language programming efficiency for "user programs" was debated before, so the same question for "systems programs" was debated after. Fortunately, the earlier debate did little to interfere with the development of high-level languages—most users were willing and anxious to use them, and computer centers were willing and anxious to satisfy the demand of the market. Unfortunately, the latter debate has hurt, seriously in some cases, the ability of third generation programming projects to meet their deadlines and to stay within their budgets.

The case for high-level languages for operating systems programming rests on four points [C3, C6]: 1) gross strategy changes can be effected with relative ease; 2) programmers may concentrate their energies and productivity on the problems being solved rather than on machine details; 3) programmers are more likely to produce readable text, an invaluable aid to initial implementation, to testing and documentation, and to maintenance; and 4) a principal source of system bugs, incompatibilities in the interfaces between modules authored by different programmers, can be removed by the more comprehensible intermodular communications provided by high-level languages. However valid might be the argument that machine code is more efficient than compiled code, especially in heavily-used parts of the operating system, it appears to miss the point: the modest improvements in efficiency that are gained using machine code simply cannot offset the cost of failing to achieve the abovementioned four advantages of high-level languages, especially during the early stages of complex projects when the design is not fixed.

The writing of operating systems programs is, however, more difficult intellectually than "ordinary applications" programming. The essential difference is: the writer of an ordinary program is concerned with directing the activity of a single process, whereas the writer of an operating system program must be concerned with the activities of many, independently-timed processes. This sets two extraordinary requirements for operating systems programming languages. First, the language must provide the ability to coordinate and control concurrent activities (though this ability is not present in most programming languages, it is present in a number of simulation languages [M1, W3]); and, given this ability, the programmer must master the faculty of thinking in terms of parallel, rather than serial, activity. Secondly, in order to permit and encourage efficient representations of tables, queues, lists, and other system data areas, the language must provide a data structure definition facility. Not only must the programmer be skilled in defining and using data structures, he must be able to evaluate which one is most efficient for a particular need. Although these two requirements must be met whether or not a high-level language is used, a suitable language can smooth the way to their realization.

Several systems—such as the Burroughs B6500 [C6], IDA's time-sharing system [I3], and MULTICS [C3]—have made successful use of high-level languages throughout their operating systems.

Procedures

Because it is intimately related to programming modularity, the procedure (subroutine) has always been an important programming concept. Recursive and shared procedures have assumed positions of particular importance since 1960. An abstract description of "procedure in execution" is useful for understanding how an operating system implements an environment for efficient execution of procedures.

A procedure is said to be *activated* (i.e., in execution) between the time it is called and the time it returns. An activated procedure consists of a set of instructions and an "activation record" (i.e., working storage). Each activation of a procedure must have a distinct activation record because, without it, computations on local or private data would be meaningless; such computations arise under recursion and sharing. The activation record defines a local environment for the procedure, containing all operands that are to be immediately accessible to instruction references. Any objects that must be accessed but are not in the activation record (e.g., certain parameters of the procedure call) must be accessed indirectly through pointers in the activation record; all such objects constitute the nonlocal environment of the procedure activation. An activation record must also contain information by which control can be returned to the caller. Since activation records need exist only when the procedure is activated,* a scheme of dynamic allocation of storage for activation records is required.

An implementation of procedures based on the foregoing ideas must solve three problems: 1) the allocation and freeing of storage for activation records at procedure call and return; 2) the efficient interpretation of local operand references; and 3) the interpretation of nonlocal operand references. The solutions to these problems will depend on the objectives of the system and on the types of languages used; therefore, we shall not attempt to specify them in detail here. Good treatments are given in [J1, W3].

For the purposes of this discussion, assume that the memory is partitioned into two contiguous areas: a (read-only) area to hold the instruction parts of all procedures, and an area to hold activation records. Each activation record is assumed to occupy a contiguous region of memory. Each procedure activation is described at any given point by the value of an *activation descriptor* (i, r), where i is the address of the next instruction to be executed and r is the base address of the activation record. As suggested in Figure 1, the processor contains a pair (I, R) of registers that hold the activation descriptor of the currently executing procedure. (The register I is the instruction counter, and R is an activation record base register.) As in Figure 1, an instruction (OP, x), specifying some operation OP that involves the contents of address x, references the xth location relative to the base of the current activation record (i.e., location $x + c(R)$, where $c(R)$ denotes the contents of R).

As mentioned above, each activation record is provided with the activation descriptor of its caller. Figure 1 shows the first cell of an activation record used for this purpose. The steps in calling a procedure are compiled into the instruction code and include: 1) obtain a free contiguous region in the data area of memory of the requisite length for the new activation record, starting at some address r'; 2) initialize the values of all cells in the activation record, and provide values of, or pointers to, the parameters of the call; 3) let r be the contents of R and i the address of the instruction to be executed after the return; store (i, r) in the first cell of the new activation record—i.e., at location r'; and 4) let i' be the initial instruction of the called procedure, load (I, R) with (i', r') and begin executing instructions. The steps in returning include: 1) release the storage occupied by the current activation record; and 2) load (I, R)

* Parts of the activation record—e.g., **own** variables in ALGOL or STATIC variables in PL/I—may have to continue their existence even after the procedure is deactivated.

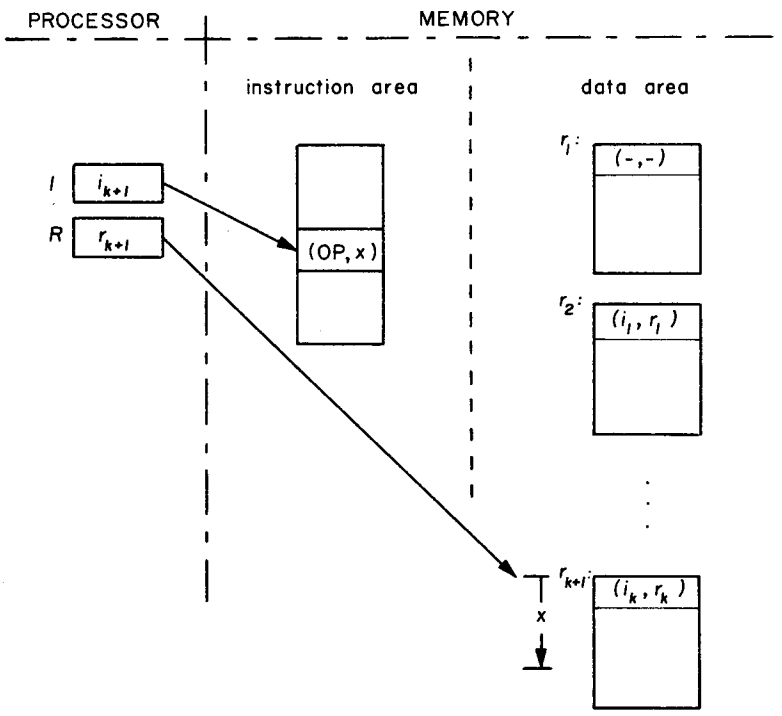


FIG. 1. Procedure implementation.

with the activation descriptor in location $r' = c(R)$.

The chain $(i_1, r_1), \cdots, (i_k, r_k)$ of activation descriptors (Figure 1) linking the activation records to their callers is known as the "dynamic chain." Since returns are executed in the reverse order of calls, many implementations handle the allocation of storage for activation records by means of a pushdown stack [W3]. It should be clear from the descriptions above that the stack mechanism is a consequence, not a prerequisite, of procedure implementation.

The most common method for handling nonlocal operand references (those to be interpreted outside the context of the current activation record) is based on the properties of block-structured languages, such as ALGOL, where blocks define a hierarchy of nested contexts. These implementations treat block entry and exit the same as procedure call and return (except for parameter passing), and provide each activation record with a pointer to the activation record of the lexicographically enclosing block in the program text. The chain of pointers so defined is known as the "static chain." Complete descriptions of the problems involved in constructing and using static chains can be found in [J1, W3].

STORAGE ALLOCATION

Of all the resources of a system, memory is perhaps the most scarce, and so techniques and policies for managing it have consistently received much attention. 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 quantities of (fast, expensive) main memory and (slow, cheap) auxiliary memory.

To understand modern memory systems, one must understand two sets of techniques, *memory management* and *name management*. The techniques of memory management concern the accessing and placement of in-

formation among two or more levels of memory. The most important among these techniques is the "one-level store," or virtual memory. The techniques of name management are concerned with achieving very general forms of programming modularity by providing for the definition of different contexts within which processes can access objects. The most important among these techniques are file systems and "segmentation."

In the following discussion, the terms *computational store* and *long-term store* will be used. The computational store is that part of the memory system in which objects must reside so that a process can reference them using the hardware addressing facilities of the system. The long-term store is that part of the memory system in which objects not immediately required in any computation may reside indefinitely. Both terms refer to memory as presented to the programmer. Both may use main and auxiliary memory in their implementations. Most systems have distinct computational and long-term stores; a few do not.

Motivations for Automatic Memory Management

Most of the modern solutions to the automatic storage allocation problem derive from the one-level store, introduced on the Atlas computer [K1]. This machine made the distinction between "address" and "location," with an address being the name for a word of information and a location being a physical site in which a word of information is stored. The set of all addresses a processor can generate as it references information (program addresses) has come to be known as *address* or *name space*, and the set of all physical main memory locations (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 programming, for the name space can be associated permanently with the program and made independent of prior assumptions about memory space. The memory management problem becomes the system's problem as it translates program addresses into hardware addresses during execution. Examples of contemporary systems are discussed in [R1].

The one-level store implements what appears to the programmer as a very large main memory without a backing store; hence, the computational store is a large, simulated (virtual) main memory. Two lines of argument are used to justify this approach: the first reasons that the "overlay problem" can be solved by simulating a large, programmable memory on a machine with relatively smaller main memory; the second traces the possible times at which program identifiers are "bound" to (associated with) physical locations, and reasons that the one-level store provides maximum flexibility in allocating memory resources by delaying binding time as long as possible.

The argument for an automated solution to the overlay problem proceeds as follows. Since main memory in the early stored-program computers was quite small by today's standards, 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 into the program text the appropriate commands to do so.* The introduction of algorithmic source languages (e.g., FORTRAN and ALGOL) and the linking loader made it possible to construct large programs with relative ease. As the complexity of programs increased, so grew the magnitude of the overlay problem; indeed, 25 to 40% of programming

* Typically, the programmer would divide his information into "objects" and his computation time into "phases." During any phase only a subset of the objects would be referenced. The problem was to choose the objects and phases such that: 1) the total space occupied by referenced objects during any phase did not exceed the size of main memory; 2) the phases were as long as possible; 3) any object remaining in memory in successive phases did not have to be relocated; and 4) the amount of information that had to be moved out of memory during one phase, to be replaced (overlaid) by information moving in for the next phase, was minimal. In addition to these requirements, it was desirable to overlap the overlaying process as much as possible with execution. For programs involving sizable numbers of phases and objects, finding any reasonable solution (much less an optimal one) to the overlay problem was a formidable task. The problem was always more severe with data objects than with procedure objects.

costs could typically be ascribed to solving the overlay problem [S3]. The need for executing large programs in small memory spaces thus motivated the development of hardware and software mechanisms for moving information automatically between main and auxiliary memory.

The argument for postponing binding time 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 addresses are bound to storage locations by the compiler; 3) if the program is specified as a collection of subroutines, program addresses are bound to storage locations by the loader. In these three cases, binding is permanent, once fixed. It may, however, also be dynamic: 4) storage is allocated (deallocated) on demand, such as at procedure activation (deactivation) times in ALGOL or at data structure creation (deletion) times in LISP and PL/I; and 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 time. Although postponing binding time increases the cost of the implementation, it does increase freedom and flexibility in constructing programs and in allocating resources.

The two preceding paragraphs outline the qualitative justification for automatic storage allocation. Further discussions of these points from different perspectives can be found in [A2, D3, D6, D7, R1]. Quantitative justification is provided by Sayre [S3]. However one arrives at the conclusion that some form of dynamic storage allocation is required, it is apparent that the programmer cannot handle it adequately himself. Not only is he not privy to enough information about machine operation to make allocation decisions efficiently, but solving the overlay problem at the programming level requires extensive outlays of his valuable time, and the results are not consistently rewarding.

Virtual Memory

Modern memory systems are studied by means of the following abstractions: address space, memory space, and address map. These abstractions allow one to discern a pattern among the seemingly endless variety of existing memory systems. They are summarized in the form 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 that 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 (Figure 2) to translate processor-generated addresses (from N) into locations (in M). This reflects our earlier requirement that the address space N be independent of prior assumptions about M; as the programmer is aware only of N and not of the two levels of memory, the processor that executes his program can generate addresses from N only.

Let us consider some salient aspects of virtual memory implementations. Diagrams more detailed than Figure 2 indicating the operation of address mapping in specific cases have been given in [D1, D6, D7, R1, W3, W6] and are not reproduced here. The address map f is normally presented in the form of a table so that it can be accessed and updated efficiently. However, 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, N is partitioned into "blocks" of contiguous addresses, and separate mapping information is provided for blocks only,* thus reducing the potential

* If b is a block number, then either $f(b)$ is the base address of a contiguous region of M containing block b, or $f(b)$ is undefined if block b is missing from M. The mapping mechanism makes a sequence of transformations, $x \rightarrow (b, w) \rightarrow (f(b),$

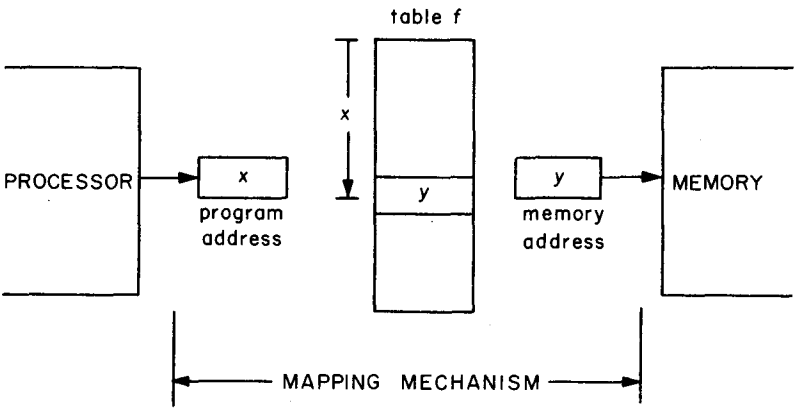


FIG. 2. Address translation.

size of the mapping table to manageable proportions.

In most implementations, the mapping table f is stored in main memory as a directly-indexed table (suggested in Figure 2), and a small associative memory is included in the mapping hardware to speed up the translation process.** With respect to the blocks themselves, there are two alternatives: the block size is either fixed and uniform or it is variable. When the block size is fixed and uniform, the blocks of address space are called "pages" and the blocks of main memory, "page frames." The second alternative arises when one considers defining the blocks of address space according to natural logical boundaries, such as subroutines. When deciding which alternative to use, the designer must take into account the efficiency of the mapping mechanism, the efficiency of storage utilization, and the efficiency of possible allocation policies [D6]. The designer must also be aware of two conflicting tendencies: on the one hand, fixed block size leads to simpler, more efficient storage management systems when properly designed, whereas the need for efficient name management requires that N be partitioned according to logical boundaries. A compromise, known as "segmentation and paging," will be discussed below.

Observe that the foregoing definition of an address map is independent of the physical realizations of main and auxiliary memory. One common type of memory system uses magnetic core main memory and drum auxiliary memory, the translation of addresses and management of memory being implemented by a combination of hardware and software [D6, R1]. Another common type of memory system uses semiconductor register main memory and magnetic core auxiliary memory, the translation of addresses and management of memory being implemented entirely in hardware [L2, W4]. The former type of memory system is the classical "core-drum" system derived from the Atlas computer; the latter is the "cache store" or "slave memory" used in many late third-generation machines (e.g., the IBM 360/85, certain members of the IBM 370 series, and the CDC 7600).

All types of virtual memory systems have the five important properties detailed below.

$w) \longrightarrow y = f(b) + w$, where x is a program address, b is the number of the block containing x, w is the address of x relative to the base of b, and y is a memory address. The transformation $x \longrightarrow (b, w)$ may be omitted if, as is sometimes the case, elements of N are already represented as (b, w) pairs [D6].

** The associative memory consists of cells containing entries of the form $(b, f(b))$. Whenever a reference to block b is made, the associative memory cells are searched in parallel for an entry $(b, f(b))$. If the required entry is found, the base address $f(b)$ of the block is available immediately; otherwise, the table f in memory must be accessed, and the entry $(b, f(b))$ replaces the least recently used entry in the associative memory. In the former case, the time required to complete the reference to the desired address-space word is one memory cycle time, while in the latter case it is two memory cycle times. Associative memories of only 16 cells have been found to bring the speed of the mapping mechanism to within 3.5% of the speed at which it would operate if the address map could be accessed in zero time [S4].

1) Virtual memory fulfills a variety of programming objectives: (a) memory management and overlays are of no concern to the programmer; (b) no prior assumptions about the memory space M need be made, and the address space N is invariant to assumptions about M; and (c) physically, M is a linear array of locations and N is linear, but contiguous program addresses need not be stored in contiguous locations because the address map provides proper address translation; thus, the address map gives "artificial contiguity" and, hence, great flexibility when decisions must be made about where information may be placed in main memory.
2) Virtual memory fulfills a variety of system design objectives in multiprogramming and time-sharing systems; the abilities to: (a) run a program partly loaded in main memory; (b) reload parts of a program in parts of memory different from those they may previously have occupied; and (c) vary the amount of storage used by a program.
3) Virtual memory provides a solution to the relocation problem. Since there is no prior relationship between N and M, it is possible to load parts of N into M without regard to their order or previous locations in M. However, it may still be necessary to use a loader to link and relocate subprograms within N.
4) Virtual memory provides memory protection: a process may reference only the information in its address space (i.e., in the range of the address map), any other information being inaccessible to it. This is an immediate consequence of the implementation of address translation, according to which each address must be mapped at the time it is generated. It is, therefore, impossible to reference any information not represented in the map. Assuming the correctness of the map, it is likewise impossible to reference unauthorized information. In addition, it is possible to include "protection keys" in each entry of the map, so that only references of the types specified by the keys (e.g., read, write, execute) will be permitted by the mapping hardware.

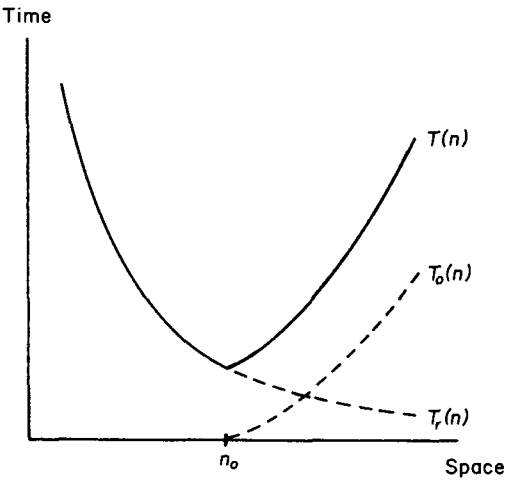


FIG. 3. Space and time in virtual memory.

5) The space/time tradeoff does not hold in a virtual memory. Let $T_r(n)$ denote the running time of the fastest program that correctly solves a given problem when the address space size is n and the entire program is loaded in main memory. The space/time tradeoff states that the best program is faster in larger memories; i.e., $T_r(n + 1) \leqslant T_r(n)$. When the main memory size is m, the observed running time is $T(n) = T_r(n) + To(n)$, where To is the overhead in the storage management mechanism; $To(n) = 0$ for $n \leqslant m$ and $To(n + 1) \geqslant To(n)$ for $n > m$ (see Figure 3). Therefore, when $n > m$, it may easily happen that $T(n + 1) > T(n)$, i.e., space and time need not trade with respect to observed running time. Since prior knowledge of m is denied the programmer, he cannot rely on the tradeoff. To achieve good performance from a virtual memory system, the programmer must instead use techniques that maximize the "locality of reference" in his program [D6, pp. 183–7].

To this point, the discussion has dealt with the mechanisms of virtual memory. Once the mechanism has been chosen, a policy is needed to manage it. A memory management policy must strive to distribute information among the levels of memory so that some specified objective is achieved. If, for example, the objective is maximum system operating speed, the policy should seek to retain in main memory the informa-

tion with the greatest likelihood of being referenced. A memory management policy comprises three subpolicies: 1) the fetch policy, which determines when a block should be moved from auxiliary memory into main, either on demand or in advance thereof; 2) the placement policy, which determines where in the unallocated region of main memory an incoming block should be placed; and 3) the replacement policy, which selects blocks to be removed from main memory and returned to auxiliary. The complexity of the three subpolicies can be compared according to whether block size is fixed or not, and the complexity of a good policy may well affect the choice of whether to use fixed or variable block size. For example, in paging systems all blocks are identical as far as the placement policy is concerned, and in demand paging systems (which fetch on demand and replace only when main memory is full) the memory management policy reduces to a replacement policy [B1, M2].

Demand paging is widely used and well documented in the literature. Many policies have been proposed and studied [B1, D6, M2]; the basis for most is the "principle of optimality" for minimizing the rate of page replacements: replace the page having the longest expected time until reuse. Although this principle is not optimal for arbitrary assumptions about program behavior, it is known to be a very good heuristic.

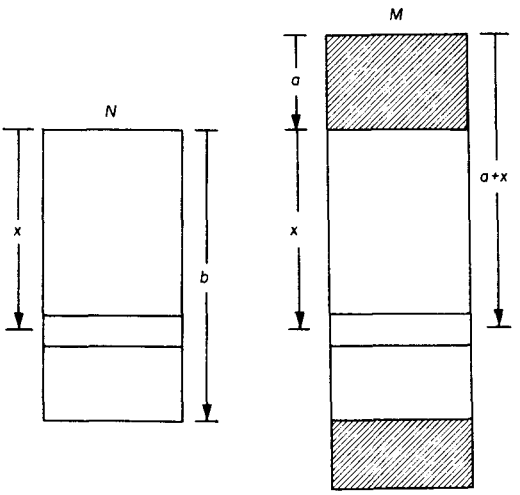


FIG. 4. Relocating an address space in memory.

The study of main memory management can be rounded out by the study of a dual problem, auxiliary memory management. Many of the considerations identical to those discussed above are relevant here: whether or not block size should be fixed, what the block size(s) should be, and how to store tables locating blocks in the auxiliary memory. However, the policies of auxiliary memory management are not the same as those of main memory management, because the objective is reducing channel congestion and waiting times in auxiliary memory queues. The "shortest-access-time-first" policies for managing request queues are known to be optimal or nearly so [A1, D6].

The auxiliary memory of many a large paging system is comprised of both drums and disks, the drums being used to swap pages of active programs to and from main memory, the disks 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 movement of pages of such files between drum and disk (usually through main memory buffers); in some systems it can be a serious problem with no obvious, simple solution [W6, p. 45].

Extending the foregoing considerations to multiprogramming is straightforward. In most cases, a separate address space and address map are associated with each process. There are two ways to map the resulting collection of address spaces into main memory. The first uses "base and bound" registers to delineate a contiguous region of memory assigned to a given address space. As shown in Figure 4, the address space N, consisting of b words, is loaded contiguously in main memory starting at base address a. The address map $f:N \rightarrow M$ is a simple translation defined by $f(x) = x + a$; it can be implemented by the mapping mechanism shown in Figure 5. The base register (known also as a relocation register) contains the address a, which is added to the address x generated by the processor. The address x can be checked against the bound b (address space size) and an error signal generated if $x > b$. When the processor is switched to a new process, the base and bound registers

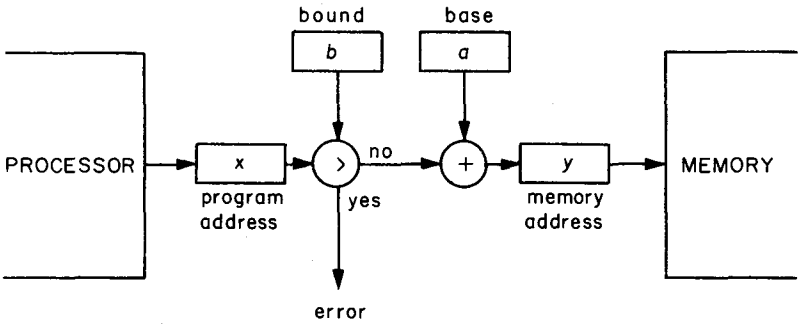


FIG. 5. Use of base and bound registers.

must be reloaded to contain the mapping information for the new process. This method is useful only when main memory can hold several address spaces, as for example in the CDC 6600 or the IBM OS/360-MVT.

The second method of mapping a collection of address spaces into main memory is a direct extension of the block-oriented mappings discussed earlier: 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. This second alternative is widely used, for it permits mapping of address spaces larger than main memory. When properly designed, it can be especially efficient under paging.

Memory management policies for multiprogramming are best understood with the aid of a *working-set* model for program behavior. A program's working set at any given time is the smallest subset of its address space that must be loaded in main memory in order to guarantee a specified level of processing efficiency. (For proper choices of the parameter T in a paging system, one can use the set $W(t, T)$ of pages referenced among the T page-references immediately preceding time t as an estimator of the working set at time t [D3, D6].) The working-set concept has implications for both the programmer and system designer. A programmer can realize good performance from a virtual memory system by creating programs whose working sets are small, stable, and slowly changing. (If this is achieved, the total amount of address space employed is immaterial.) A system designer can realize good performance under multiprogramming by designing policies which guarantee that each active process has its working set loaded in main memory. If this principle is not followed, attempted overcommitment of main memory may induce collapse of performance, known as thrashing [D4]. The performance of core/drum paging systems can be improved by departing from strict adherence to demand paging: during a time slice, pages can be added on demand to the working set of a process and replaced when they leave it; but at the beginning and end of a time slice, swapping can be used to move the working set as a unit between main and auxiliary memory.

Motivations for Name Management

The techniques of virtual memory outlined above define a computational store which appears to the programmer as a very large, linear name space. But, once presented with a single linear name space, no matter how large, the programmer is still faced with the need to solve four important problems:

1) handling growing or shrinking objects, and creating or deleting objects, without having to resort to using overlays in name space or to introducing dynamic storage allocation procedures into name space (in addition to the ones implementing the computational store);
2) providing long-term storage and retrieval for named information objects;
3) allowing information to be shared and protected; and
4) achieving programming modularity, according to which one module can be compiled (into some part of name space) with-

out knowledge of the internal structure of other modules and without having to be recompiled when other modules are.

A complete discussion of why linear name space fails to satisfy these objectives has been given by Dennis [D7]. The basic problem is that the dynamics of information additions and deletions in name space implied by each of these objectives require some method of structuring and managing names, a method not intrinsically provided by the virtual memory mechanisms as described above.

Modern computer systems use two methods for achieving these objectives: 1) *file systems*, by far the most widely used, which provide a storage system external to the name space; and 2) *segmented name space*, used in MULTICS [B2], which alters the structure of the name space itself. The file system implements a long-term store separate from the computational store, while the segmented name space implements a long-term store identical to the computational store. We shall comment on each below.

File Systems

Many of today's operating systems provide both a virtual memory and a file system, an apparent paradox since the virtual memory is supposed to conceal the existence of auxiliary memory, while the file system does not. The paradox is resolved in the light of the inability of linear name space to satisfy the four objectives given above. The programmer is presented with a pair of systems (N, F), where N is a linear name space and F is the file system. An address map $f: N \rightarrow M$ is implemented in the usual way. The file system consists of a *directory* or *directory tree* [D2, D11] and a set of *file processing primitives*. The directory represents a mapping from file names to auxiliary memory. Arranging a collection of directories in a tree has two advantages: 1) each directory can be identified uniquely by its *pathname*, i.e., the sequence of directory-names on the path from the root of the tree to the given directory; and 2) the same name can be reused in different directories. Observe that the directory tree can be regarded as a "file name space" in which objects (files) are named by pathnames; this is contrasted with the linear name space discussed earlier. In addition, a directory tree serves to provide various contexts within which processes may operate; it does this by allowing each process to have a pointer designating a "current directory" in the tree. The file processing primitives include operations like "search directory"; "change context to predecessor or successor directory"; "link to an entry in another directory"; "open file"; "close file"; "create file"; "delete file"; "read file into N"; and "write file from N." The directory trees of individual users can be made subtrees of a single, system-wide directory tree, and users can employ the linking primitive to share files. Protection keys can be stored in directory entries. Since files can grow and shrink, be created and destroyed, be stored for long periods, be shared and protected, and be program modules, the four programming objectives given above are satisfied.

Yet, file systems have an important limitation, which is a consequence of the distinction between computational and long-term storage systems: in order to take advantage of the efficiencies of the addressing hardware, the programmer is forced to copy information between the file system and the computational store; this, in effect, reintroduces a problem that virtual memory sought to eliminate—the overlay problem in name space. As will be discussed shortly, the segmented name space overcomes this problem by removing the distinction between the computational and long-term storage systems.

Frequently mentioned in discussions about file systems is "file structure," i.e., the organization of information within a file [I1]. Three types of file structures are in common use:

1) bit or word strings (a linear address subspace);
2) chains of records; and
3) a collection of records, each having a unique index (key), with the indexes having a natural order; a given record can be located using hashing techniques when its index is presented, or the records can be searched sequentially in the order of the indexes.

The first structure represents a fundamental way of thinking about files; the second and third do not. The physical properties of auxiliary memories (disks, drums, tapes) have always forced the use of records in the implementation of a file; but, just as the blocks of memory used by the mapping mechanism need not be visible to the programmer, so the records of a tape or the tracks of a disk or drum need not be visible. In other words, records and sequential indexing are not prerequisites to understanding a file system. By confusing the implementation of mappings with the logical properties of files, many systems present a confusing and overly complicated picture to the programmer in this respect.

Segmentation

The power of the segmented name space to solve the four programming problems presented under "Motivations for Name Management," above, without the limitations to which file systems are subject, appears not to be widely appreciated. One can regard a file system as providing a collection of named, linear address subspaces (files) of various sizes, augmenting the main name space implemented by the computational store. From this, we can make the generalization that, instead, the computational store itself comprises a collection of named, linear subspaces of various sizes. Each of these subspaces is called a *segment;* the entire name space is called a *segmented name space;* and the capability for a segmented name space is called *segmentation.* Each word is referenced by a two-dimensional address (segment name, word name); this is contrasted with linear name space in which a single dimension of addressing is used. Since all segments may reside permanently in the name space (no separate file system is required) and the programmer does not have to program file operations or solve an overlay problem, the four programming problems are solved.

Under segmentation, however, the system must handle the linking of segments into a computation, and special hardware and compiling techniques are required to make this process efficient [D1]. (In systems without segmentation, the programmer implicitly solves the linking problem while copying files from the file system into specified areas of address space.) The most commonly proposed approach to sharing segments among the name spaces of distinct processes uses a system-wide directory tree, as discussed above for file systems [B2, D2], so that each segment is represented twice in the system.

There are two methods for mapping a segmented name space into main memory. The first stores a segment contiguously in main memory, in the manner of Figures 4 and 5, except that the base-and-bound information is stored in the mapping table. This technique is subject to the same limitations as the variable-block-size mappings discussed earlier [D6]. The second method, known as *segmentation and paging* [D7], uses paging to map each segment, which is a linear name space in its own right. It uses two levels of tables to map a given segment-word pair (s, x) to a main memory location y in the following steps: 1) the segment name s is used to index a "segment table" whose sth entry gives the base address of a "page table" for segment s; 2) the word name x is converted to a pair (p, w), where p is a page number and w the relative address of x in page p; 3) the page number p is used to index the page table whose pth entry contains the base address q of the page frame holding page p; and 4) the memory address is $y = q + w$. Experiments on the MULTICS system demonstrate that the speed degradation caused by using this mapping technique, together with a 16-cell associative memory, averages 3.5% [S4]; thus, segmentation and paging competes in efficiency with more standard forms of virtual memory.

CONCURRENT PROCESSES

We mentioned earlier that concurrency is an important aspect of computer systems, that mechanisms for controlling it must be available for use in some programming languages, and that programmers must understand the basic concurrency problems and their solutions. Even though the present hardware technology allows for the realization of con-

current activity, it remains the task of the programmer to write the programs that bring it into being.

It appears more natural to regard a computer system as a set of processes cooperating and rendering service to one another, rather than as a set of subroutines performing in the same fashion. In other words, the unit of system decomposition is the process, not the subroutine [B5]. This view pervades the design of Dijkstra's THE multiprogramming system [D14] and has influenced other third generation systems, though not as strongly. Under this view, the most elementary conceivable operating system is one that provides for the creation, removal, control, and intercommunication of processes; all other operating system functions can be organized around this [B5].

The concept "process" appears to have originated in several design projects during the early 1960s; for example, in MULTICS and in OS/360 (where it is called a "task"). It was intended as an abstraction of the activity of a processing unit, and was needed to solve the problem of preempting and restarting programs without affecting the results of their computations. Although many early definitions of "process" were imprecise, they were adequate to enable system designers to implement multitasking. Some examples of early definitions are: "locus of control in an instruction sequence" [D11]; "program in execution on a pseudo-processor" [S1]; "clerk carrying out the steps of an algorithm" [V1]; and "sequence of states of a program" [H5]. The intent of these definitions is to distinguish the static program from the dynamic process it generates, and to suggest that the term "program" be applied to a sequence of instructions, while the term "process" be used to denote a sequence of actions performed by a program.

A precise definition of process can be developed as follows. With a given program we associate a sequence of "actions" $a_1a_2 \cdots a_k \cdots$ and a sequence of "states" $s_0s_1 \cdots s_k \cdots$. For $k \geq 1$, action a_k is a function of s_{k-1}, and s_k is the result of action a_k. The states represent conditions of a program's progress, with s_0 the initial state. The sequence $a_1a_2 \cdots a_k \cdots$ is called an *action sequence*, and $s_0s_1 \cdots s_k \cdots$ a *computation*. A *process* is defined as a pair $P = (s, p)$, where p is a program and s is a state. The process (s, p) implies a future action sequence $a_1a_2 \cdots a_k \cdots$ and a computation $s_0s_1 \cdots s_k \cdots$, where $s_0 = s$. Interrupting a process (s, p) after k steps is equivalent to defining the new process (s_k, p), and (s_k, p) is called a "continuation" of (s, p). We have defined "process" in this way in order to capture the flavor of its usage in practice, where a process (s, p) can be implemented as a data structure and be regarded as an entity that demands resources. Neither the action nor the state sequences are by themselves adequate for this.

The interpretation of "action" and "state" depends on the process control problem under consideration. For example, if we were interested in preempting processor or memory resources under multiprogramming, a "state" would consist of a specification (or implied specification) of the contents of all processor registers and name space locations, and an "action" would be an instruction execution. Thus, an adequate description of the process for program p is the vector (i, R, f, p), in which i is the instruction counter, R is the contents of all processor registers, and f is (a pointer to) the address map.* Preempting this process requires saving the information (i, R, f). Other interpretations of "action" and "state" are discussed below.

In order to apply the foregoing concepts to a system, we need to generalize them. To do this we define a *system of processes* to be a collection of processes, together with a specification of precedence constraints among them. If process P_1 precedes process P_2, then P_1 must terminate execution before P_2 can begin. The precedence constraints are assumed to be physically realizable in the sense that no process can precede itself. (In mathematical terms, the precedence constraints partially order the system of processes.) Two processes without any prece-

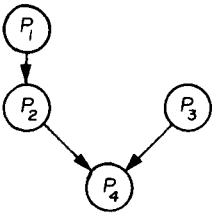
* Note the similarity between this definition of "process" and the activation descriptor definition given above for a procedure in execution. In fact, the definition of "process" is a direct generalization of that of "procedure in execution."

dence constraint between them are *independent*, and may be executed in parallel. An action sequence generated by the system consists of concatenations and merges of the action sequences of its component processes, consistent with the precedence constraints; that is, a system action sequence is obtained by merging the process action sequences such that, if P_1 precedes P_2, then the actions of P_1 must precede those of P_2 in the system action sequence. Even though the process action sequences are unique and the precedence constraints fixed, there may be many possible system action sequences, depending on the total number of possible ways of merging the sequences of independent processes. Stated differently, the action sequence observed in a given run of a system of processes will depend on the relative speeds of the individual processes, which, in general, may not be predictable. Figure 6 illustrates a system of four processes and three possible system action sequences. Finally, of course, each possible system action sequence will correspond to a sequence of system states, i.e., to a system computation.

Let $P_i = (s_i, p_i)$ be the ith process in a given system. If s_i is not the final state of P_i (i.e., if further actions are possible by P_i), then any process P_j that P_i precedes is *blocked* or *disabled*. If P_i is not blocked and s_i is not its final state, then P_i is *active*.

With respect to a system of processes, there are four control problems of importance.

1) The *determinacy* problem arises whenever a system of processes shares memory cells. The system is determinate if, no matter which system action sequence arises, the result of the computation, as manifested in the contents of memory cells, depends uniquely on the initial contents of memory cells. In other words, the outcome of the computation is independent of the relative speeds of the processes in the system.
2) The *deadlock* problem arises in limited resource systems where it may be possible for a subset of the processes to be blocked permanently because each holds nonpreemptible resources requested by other processes in the subset.
3) The *mutual exclusion* problem arises whenever two or more independent processes may require the use of a reusable resource R. Since each process modifies the internal state of R while using it, no two processes may be using R at the same time. Since each process initializes the internal state of R when it first acquires it, the order in which processes gain access to



Precedence Diagram

Process	Action Sequence
P_1	$a_1 b_1 c_1$
P_2	$a_2 b_2$
P_3	$a_3 b_3 c_3$
P_4	$a_4 b_4$

Sample System Action Sequences:

$a_1\ b_1\ c_1\ a_2\ b_2\ a_3\ b_3\ c_3\ a_4\ b_4$

$a_1\ b_1\ c_1\ a_3\ b_3\ c_3\ a_2\ b_2\ a_4\ b_4$

$a_1\ b_1\ a_3\ c_1\ b_3\ a_2\ c_3\ b_2\ a_4\ b_4$

FIG. 6. A system of processes.

R is immaterial. Two processes are mutually excluded with respect to R if at most one may be using R at a time.

4) The *synchronization* problem between two processes arises whenever the progress of one depends on that of the other. Common examples of synchronization include: the requirement that one process cannot continue past a certain point until a signal is received from another; and, in the case of certain cooperating cyclic processes, the requirement that the number of completed cycles of one should not differ from the number of completed cycles of the other by more than a fixed amount.

These four control problems are described in greater detail below. Their solutions constrain the system of processes, either in the design or by the addition of external control mechanisms, so that only the desirable system action sequences and computations arise. The solutions given here should be regarded as models for the desired behavior of systems; they do not necessarily represent the most practical approach for a specific problem. The designer would have to be prepared to modify the solutions to suit the exigencies of the system at hand. Discussions of other approaches can be found in [A3, A5].

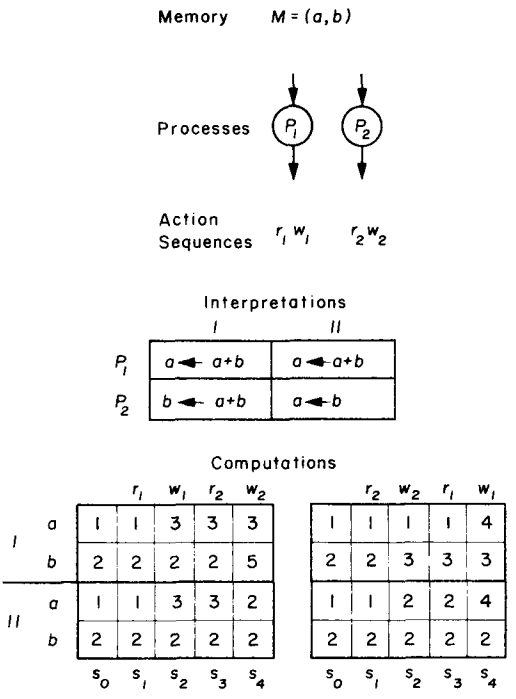


FIG. 7. Nondeterminate system of processes.

Determinacy

Consider a system of processes having access to a set M of memory cells. Each process has associated with it fixed sets of *input* and *output* cells, both subsets of M. The actions performed by each process are read or write actions on the input and output cells, respectively, of that process. A state in this system is a vector of values of cells in M; a read action leaves the state unchanged, whereas a write action may modify the values in the output cells of the process. A system computation is the sequence of such states generated by a system action sequence.

An *interpretation* for a process is a specification of the algorithm performed by the program of that process, subject to the requirement that the input and output cells of the process be fixed. A system of processes is determinate if, for all possible interpretations of the processes in the system, the final contents of M depend uniquely on the initial contents of M, regardless of which system action sequence arises. (That is, the final state of a system computation is uniquely determined by the initial state.) Since a process generates a computation depending only on the initial contents of its input cells, individual processes are determinate.

Figure 7 shows a simple example of a nondeterminate system. The two processes P_1 and P_2 are independent and use two memory cells. For $i = 1, 2$, the action sequence of P_i is $r_i w_i$, where r_i denotes a read action on the input cells of P_i and w_i denotes a write action on the output cells of P_i. Two possible interpretations, I and II, are given. For each interpretation, two computations are shown in the table, the rows being the contents of the indicated memory cells after successive actions in the action sequence, and the columns being the system states. For both interpretations, the final system state depends on the order in which the two processes perform their read and write operations; hence, the system is non-

determinate under both interpretations. It is not hard to see that interpretation I is nondeterminate because each process writes into the input cells of the other, and that interpretation II is nondeterminate because both processes attempt to write a value into the same output cell.

Two processes are *noninterfering* if: 1) one precedes the other in the system of processes; or 2) they are independent and no output cell of one is an input or output cell of the other. It can be shown that noninterference is a sufficient condition for the determinacy of a system of processes [A6]. Under general conditions, noninterference is a necessary condition for determinacy.

Deadlocks

A deadlock is a logical problem arising in multitask systems in which processes can hold resources while requesting additional ones, and in which the total demand may exceed the system's capacity even though the demand of each individual process is within system capacity. If at some point in time there is a set of processes holding resources, none of whose pending requests can be satisfied from the available resources, a deadlock exists, and the processes in this set are deadlocked.

Figure 8 is an informal illustration of the essential features of the deadlock problem [C1]. The figure shows a two-dimensional "progress space" representing the joint progress (measured in number of instructions completed) of two processes. Any sequence of points, (x_1y_1), (x_2y_2), $\cdots$, (x_ky_k), $\cdots$, starting at the origin (0, 0) in this space, in which each point is obtained from its predecessor by a unit increment in one coordinate, is called a "joint progress path." Such a path may never decrease in either coordinate because progress is assumed to be irreversible. Now, the system has one unit each of resource types R_1 and R_2. Since each process requires exclusive control over R_1 or R_2 during certain stages of its progress, no joint progress path may pass through the region of progress space in which the joint demand of the processes exceeds the capacity in either R_1 or R_2. This is called the *infeasible region*. Deadlock is possible in this system

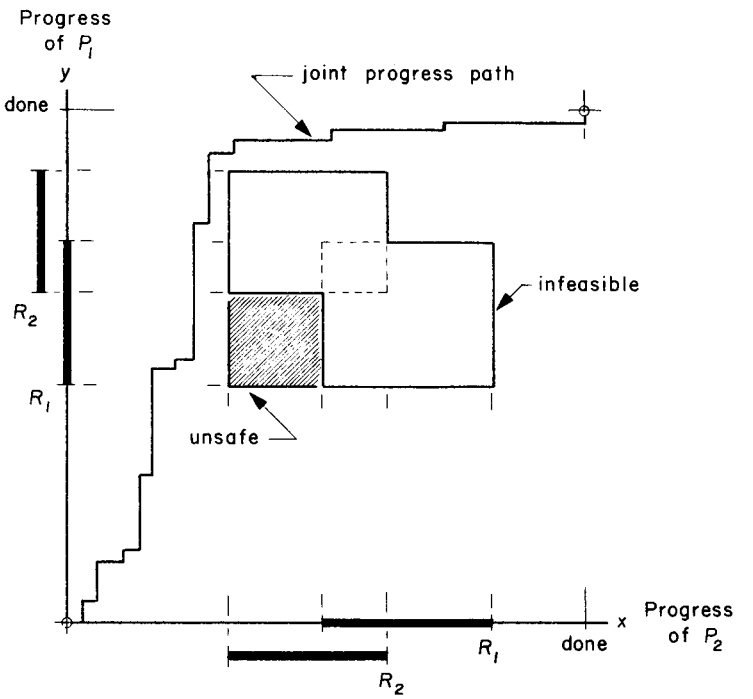


FIG. 8. Illustration of deadlock problem.

because, if joint progress ever enters the unsafe region, there is no way to avoid the infeasible region without violating the assumption of irreversibility of progress. A deadlock will exist when joint progress reaches the point at the upper right corner of the unsafe region. At that time, P_1 will be holding R_1 and requesting R_2, while P_2 will be holding R_2 and requesting R_1.

From this example one can see that there are three conditions necessary for deadlock to occur:

1) each process can claim *exclusive control* over the resources it holds;
2) each resource is *nonpreemptible*, i.e., it can be released only by the process holding it; and
3) there is a *circular wait*, each process holding a resource of one type and requesting the resource of the other type.

If a system can be designed such that any one of these conditions is precluded a priori, the system will be free from deadlock. It is not always possible to prevent deadlocks by denying the first two conditions because of various properties of resources. The third condition can, however, be precluded by a rather interesting method called "ordered resource usage": the resource types $R_1, \cdots, R_m$ are ordered and requests are constrained so that, whenever a process holds R_i and requests R_j, then $i < j$ [H2]. Restricting requests in this way is possible in assembly-line type systems (e.g., in input-execute-output programs) where a process progresses upward in the resource ordering, but never re-requests a resource type it holds or once held.

The more difficult aspects of the deadlock problem concern the detection of deadlock and the avoidance of unsafe regions in progress space. To study them, it is necessary to formalize the problem, modeling the way in which processes request and release resources. For the system-of-processes model, it is sufficient to study a system of independent processes, as deadlock can arise only among processes simultaneously competing for resources. The actions performed by a process are "requests" and "releases" of resources. A system state is a specification of: 1) how much of each resource type is currently allocated to each process; and 2) how much of each resource type is being requested by each process. The space of all possible system states constitutes a progress space analogous to that described above, and an action sequence of the system of processes generates a joint progress path. As before, there will be infeasible and unsafe regions. Coffman et al. have shown how to represent requests and releases by vectors, and system states by matrices [C1]. Holt has developed algorithms that examine a system state and decide whether or not it contains a deadlock, in a time linearly proportional to the number of competing processes [H4].

The most difficult aspect of the deadlock problem is the construction of algorithms that decide whether or not a given system state is safe; such algorithms would be useful in deciding whether granting the request of some process would put the system into an unsafe state. This problem is complicated by the requirement that some information about future resource usage of processes be available. Habermann has developed an algorithm for deciding the safeness of a state in a time linearly proportional to the number of processes, under the practical assumption that the maximum possible resource claim for each process is known in advance [H1].

Deadlock detection algorithms are used in a few contemporary systems, primarily in conjunction with peripheral device assignments [H2]. Since deadlock avoidance algorithms are a recent development, they have not yet come into wide use.

Mutual Exclusion

We stated earlier that some external control mechanism may be required to implement mutual exclusion, i.e., the requirement that, at most, one process have access to a given resource R at any one time, the order in which processes use R being immaterial. Examples of resources subject to mutual exclusion include: processors, pages of memory, peripheral devices, and writable data bases. Mutual exclusion of processors, memory, and other peripheral devices is normally provided by the system's multiplexing

mechanisms. However, there are many programming problems in which two programs may contain *critical sections* of code, i.e., sequences of instructions that must be performed by at most one program at a time because they reference and modify the state of some resource or data base [D12]. In these cases, a mechanism is required for implementing mutual exclusion at the software level.

The simplest such mechanism is provided by memory "lockout" instructions. For a given address x, this mechanism takes the form of machine instructions **lock** x and **unlock** x, whose operations are:

lock: [**if** $x = 1$ **then** $x \leftarrow 0$ **else goto lock**]
unlock: [$x \leftarrow 1$]

The brackets indicate that the enclosed actions are *indivisible*; i.e., attempting to execute more than one **lock** or **unlock** at the same time has the same effect as executing them in some order, one at a time. If the actions were not indivisible, two processors could, for example, find $x = 1$ and pass the **lock** instruction, whereas the intended effect is that only one should pass. It is important to note that the successful operation of these instructions normally depends on the fact that memory-addressing hardware arbitrates accesses to the same location, permitting one reference per memory cycle. The mutual exclusion already implemented at the lower (hardware) level is made available by the **lock** and **unlock** instructions for implementation at the higher (software) level.

To implement mutual exclusion with respect to some critical section using some resource R, a memory cell x initially containing value 1 is allocated; the programming is:

In Process P_i	In Process P_j
⋮	⋮
lock x;	**lock** x;
critical section;	critical section;
unlock x;	**unlock** x;
⋮	⋮

If both P_i and P_j attempt to access the critical section together, one will succeed in setting $x = 0$ and the other will loop on the **lock** instruction; therefore, at most one of P_i and P_j can be positioned between the instructions **lock** and **unlock.** This solution generalizes immediately to more than two processes.

Even though this is a correct solution to the mutual exclusion problem, it has three limitations. First, a processor cannot be interrupted while performing a critical section, else the variable x will remain locked and other processes may be denied the use of R for an indefinite period. Accordingly, many implementations of **lock** disable interrupts while **unlock** enables them, and some time limit on the duration of a critical section is imposed lest the processor remain uninterruptible for an indefinite period. Secondly, the solution uses the *busy form of waiting* [D13]; i.e., a processor may loop on the **lock** instruction. A more desirable mechanism would allow the processor to be preempted and reassigned. Thirdly, the solution is not "safe" if programs are cyclic; i.e., it is possible for one processor looping on the **lock** to be blocked indefinitely from entering the critical section because a second processor looping through its program may pass the **lock** instruction arbitrarily often. For these reasons the **lock** and **unlock** instructions are not normally available to users. Instead, a generalized form of **lock** and **unlock** are used, and implemented as operating system routines called by interrupts. Although the generalized forms solve the second and third problems, a timer is still required to deal with the first.

Dijkstra has defined generalized **lock** and **unlock** using the concept of *semaphore*, a variable used in interprocess signaling [D13]. A semaphore is an integer variable s with an initial value $s_0 \geqslant 0$ assigned on creation; associated with it is a queue Q_s, in which are placed the identifiers of processes waiting for the semaphore to be "unlocked." Two indivisible operations are defined on a semaphore s:*

wait s: [$s \leftarrow s - 1$; if $s < 0$ the caller places himself in the queue Q_s, enters the waiting state, and releases the processor]

* Dijkstra uses P and V for **wait** and **signal**, respectively. (The more descriptive names used here have been suggested by P. Brinch Hansen and A. N. Habermann.) OS/360 uses the ENQ and DEQ macros, which operate on the queue rather than the semaphore.

signal s: [$s \leftarrow s + 1$; if $s \leqslant 0$ remove some process from Q_s and add it to the work queue of the processors]

Semaphore values may not be inspected except as part of the **wait** and **signal** operations. If $s < 0$, then $-s$ is the number of processes waiting in the queue Q_s. Executing **wait** when $s > 0$ does not delay the caller, but executing **wait** when $s \leqslant 0$ does, until another process executes a corresponding **signal.** Executing **signal** does not delay the caller. The programming for mutual exclusion using **wait** and **signal** is the same as for **lock** and **unlock,** with $x_0 = 1$ (**wait** replaces **lock**, and **signal** replaces **unlock**). By definition, the second problem with **lock** and **unlock** is solved by **wait** and **signal**; the third problem can be solved by using a first-in-first-out discipline in queue Q_s.

Synchronization

In a computation performed by cooperating processes, certain processes may not continue their progress until information has been supplied by others. In other words, although program-executions proceed asynchronously, there may be a requirement that certain program-executions be ordered in time. This is called synchronization. The precedence constraints existing among processes in a system express the requirement for synchronization. Mutual exclusion is a form of synchronization in the sense that one process may be blocked until a signal is received from another. The **wait** and **signal** operations, which can be used to express all forms of synchronizations, are often called *synchronizing primitives.*

An interesting and important application of synchronization arises in conjunction with cooperating cyclic processes. An example made famous by Dijkstra [D13] is the "producer/consumer" problem, an abstraction of the input/output problem. Two cyclic processes, the producer and the consumer, share a buffer of $n > 0$ cells; the producer places items there for later use by the consumer. The producer might, for example, be a process that generates output one line at a time, and the consumer a process that operates the line printer. The producer must be blocked from attempting to deposit an item into a full buffer, while the consumer must be blocked from attempting to remove an item from an empty buffer. Ignoring the details of producing, depositing, removing, and consuming items, and concentrating solely on synchronizing the two processes with respect to the conditions "buffer full" and "buffer empty," we arrive at the following abstract description of what is required. Let $a_1a_2 \cdots a_k \cdots$ be a system action sequence for the system consisting of the producer and consumer processes. Let $p(k)$ denote the number of times the producer has deposited an item among the actions $a_1a_2 \cdots a_k$, and let $c(k)$ denote the number of times the consumer has removed an item from among the actions $a_1a_2 \cdots a_k$. It is required that

$$0 \leqslant p(k) - c(k) \leqslant n \qquad (1)$$

for all k. The programming that implements the required synchronization (Eq. 1) is given below; x and y are semaphores with initial values $x_0 = 0$ and $y_0 = n$:

```
pro: produce item;          con: wait x;
     wait y;                     remove item;
     deposit item;               signal y;
     signal x;                   consume item;
     goto pro;                   goto con;
```

To prove that Eq. 1 holds for these processes, suppose otherwise. Then either $c(k) > p(k)$ or $p(k) > c(k) + n$. However, $c(k) > p(k)$ is impossible since it implies that the number of completed **wait** x exceeds the number of completed **signal** x, thus contradicting $x_0 = 0$. Similarly, $p(k) > c(k) + n$ is also impossible since it implies that the number of completed **wait** y exceeds by more than n the number of completed **signal** y, thus contradicting $y_0 = n$.

Another application of synchronization is the familiar "ready-acknowledge" form of signaling, as used in sending a message and waiting for a reply [B5]. Define the semaphores r and a with initial values $r_0 = a_0 = 0$; the programming is of the form:

```
IN THE SENDER                IN THE RECEIVER
      ⋮                            ⋮
generate message;            wait r;
signal r;                    obtain message;
wait a;                      generate reply;
obtain reply;                signal a;
      ⋮                            ⋮
```

The synchronizing primitives can be used to implement the precedence constraints among the members of a system of processes. Whenever P_i precedes P_j in a system, we may define a semaphore x_{ij} with initial value 0, suffix to the program of P_i the instruction **signal** x_{ij}, and prefix to the program of P_j the instruction **wait** x_{ij}. (Implementations with fewer semaphores can be found, but this one is easiest to explain.) Letting S_1, S_2, S_3, and S_4 denote the statements of the programs of the four processes in Figure 6, the system of Figure 6 can be implemented as follows:

P_1 : **begin**; S_1; **signal** x_{12} ; **end**
P_2 : **begin**; **wait** x_{12}; S_2 ; **signal** x_{24} ; **end**
P_3 : **begin**; S_3 ; **signal** x_{34} ; **end**
P_4 : **begin**; **wait** x_{24} ; **wait** x_{34} ; S_4 ; **end**

As a final example, let us consider how the synchronizing primitives can be used to describe the operation of an interrupt system. Typically, the interrupt hardware contains a set of pairs of flipflops, each pair consisting of a "mask flipflop" and an "interrupt flipflop." The states of the flipflops in the ith pair are denoted by m_i and x_i, respectively. The ith interrupt is said to be "disabled" (masked off) if $m_i = 0$, and "enabled" if $m_i = 1$. When the hardware senses the occurrence of the ith exceptional condition C_i, it attempts to set $x_i = 1$; if $m_i = 0$, the setting of x_i is delayed until $m_i = 1$. The setting of x_i is supposed to awaken the ith "interrupt-handler process" H_i, in order to act on the condition C_i. By regarding m_i and x_i as hardware semaphores with initial values $m_i = 1$ and $x_i = 0$, we can describe the foregoing activities as an interprocess signaling problem:

IN HARDWARE

C_i occurs: **wait** m_i ;
signal x_i ;
signal m_i ;
disable: **wait** m_i ;
enable: **signal** m_i ;

IN INTERRUPT HANDLER H_i

start: **wait** x_i ;
process interrupt;
goto start;

The foregoing is, of course, intended more as an illustration of the interprocess signaling aspects of interrupt systems than as an accurate description of interrupt hardware operation.

RESOURCE ALLOCATION

The purpose of automatic (system-controlled) resource allocation is to regulate resource usage by processes in order to optimize given measures of system efficiency and good service throughout the user community. Though it is possible to develop the mechanisms of resource sharing first and the policies for using them later, desirable policies may require special mechanisms. In other words, the system designer must know which general class of resource allocation policies will be used before he specifies the final choice of mechanism.

Resource allocation policies are generally of two kinds: short-term policies, which are for making decisions on a time scale comparable to or slower than human reaction times, and long-term policies. Long-term policies tend to be used for economic considerations, such questions as: "How does one forecast the demand on the system?" "How does one charge for resource usage?" and "What is the role of pricing policy in controlling demand?" Pricing policies have been found to be important as a tool for rationing resources and as a means for controlling the tendency for users to develop countermeasures that render short-term policies ineffective [C2]. Since long-term policies are of more concern to system administrators than to system designers, we shall not pursue them further here. Good discussions of the various issues can be found in [M4, N1, S5, S6, S7, W8].

The term "automatic resource allocation" refers to the study and implementation of short-term policies within the operating system. Automatic resource allocation is necessitated—and complicated—by various factors, including:

1) system designers' consistent endeavors to relieve programmers of the burdens of resource allocation by transferring those burdens to the operating system;
2) the need to control the interference and

interaction among processes that results from the sharing of equipment by concurrent processes;
3) multitasking, which exposes the system to the danger of deadlock; and
4) the need to regulate the competition for resources because the instantaneous total demand of the user community is time-varying, subject to random influences, and may exceed the supply of resources, thus interfering with overall system performance.

These factors are the motivation for a theme to be developed in the subsequent discussion: management of processors, memory, and other resources cannot be handled by separate policies; they are aspects of a single, large resource allocation problem.

The goals of short-term policies are of two distinct and often conflicting types: efficiency and user satisfaction. The former include measures of throughput, resource utilization, and congestion on information-flow paths; the latter include measures of response time, turnaround time, and funds expended to run a job. Figure 9 illustrates the conflict between a measure of user satisfaction (waiting time) and a measure of efficiency (fraction of time a processor is idle) such as might be encountered in a simple, monoprogrammed time-sharing system. The conflict follows from a result in queueing theory, which asserts: the more likely it is that the processor queue is nonempty (and therefore the processor is not idle), the more likely it is that the queue is long (and therefore a user must wait longer for service). (Indeed, if α is the idleness of the processor, the expected waiting time in queue is proportional to $(1 - \alpha)/\alpha$ [M4].)

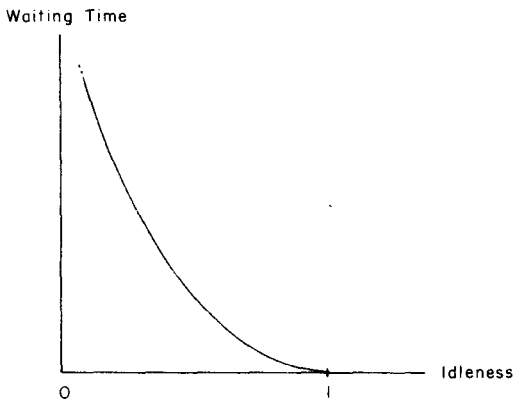


FIG. 9. A conflict between satisfaction and efficiency.

There may be other sources of conflict than service objectives. For example, in systems where deadlock avoidance routines are used, an allocation state is deemed "safe" if there exists a schedule ordering resource requests and releases of processes such that all may be run to completion without deadlock. The designer must be careful that such schedules are consistent with any pre-existing priority rules; otherwise, a deadlock may be generated by a conflict between the order of processes specified in the deadlock-avoidance schedule and the order specified by the priorities.

A Resource Allocation Model

Most systems are organized so that, at any given time, a process will be in one of three *demand-states*: 1) in the *ready* state it is demanding, but not receiving the use of, processor and memory resources; 2) in the *active* state the working set of the process resides in main memory; and 3) in the *blocked* state it is not demanding use of processor or memory, but is either demanding to use peripheral equipment or waiting for a message from another process. The allowable transitions among the states are shown in Figure 10. These demand-states are reflected in the organization of queues in computer systems, giving rise to a network of queues with feedback. In general, there are three subnetworks corresponding to each of the three states: 1) ready processes are distributed among "levels" of a queue, each level corresponding to a prediction of future resource usage; 2) active processes may be partitioned into "young" and "old," depending upon whether they were recently initiated or not (they may also be partitioned into executing and page-waiting processes in a virtual memory system); and 3) blocked processes may be partitioned according to the reasons for their being blocked. In all three cases, the subdivisions within a given class reflect various demand substates of interest.

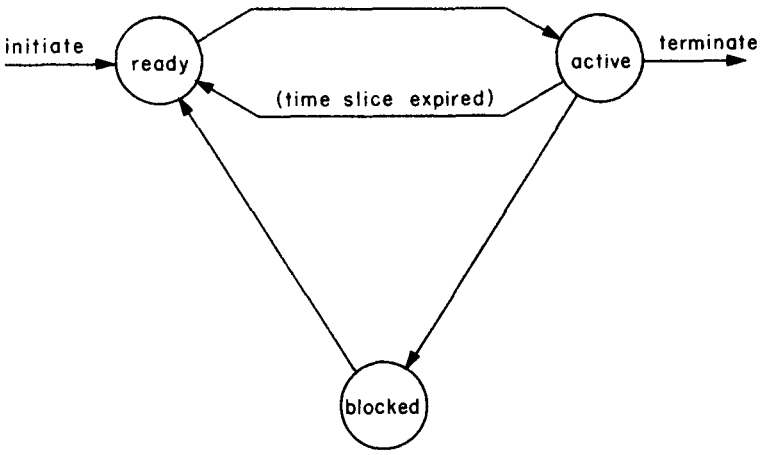


FIG. 10. Process demand-states.

The extensive literature on "scheduling algorithms," accumulated before multiprogramming became predominant, ignores the blocked state of Figure 10. Good surveys of these methods are found in [C2, M4], and a unified view of their operation in [K3]. These early models have been found to be of limited applicability in multiprogramming systems since they treat systems with only a single point of congestion (e.g., processor), whereas Figure 10 implies a system with many points of congestion (e.g., auxiliary devices, main memory, console input/output routines, etc.). In fact, memory management is often more critical than processor scheduling because processors may be switched among processes much more rapidly than information among memory levels. Thus, the applicability of these models is limited by their failure to reflect accurately job flow and system structure, not by their frequent use of exponential assumptions in analyses [F1]. But despite their shortcomings, they have made an undeniable contribution to our present understanding of resource allocation.

Successful analyses of queueing networks representative of Figure 10 have demonstrated their applicability to modern systems, and have shown that resource allocation policies may be regarded as job-flow regulation policies [B6, K2]. This approach promises to yield insight into optimal policies.

The working-set model for program behavior can be used to gain insight into those aspects of the resource allocation problem dealing with the flow between the ready and active states. It illustrates what appears to be a manifestation of a more general principle: there may exist situations in which a process must be allocated a critical amount of one resource type before it can use any resources effectively. Specifically, a process cannot be expected to use a processor efficiently if its working set is not loaded in main memory. This fact has two implications with respect to multiprogramming. First, attempted overcommitment of main memory in core-drum systems may result in the overall collapse of system processing efficiency, which is known as thrashing [D4, D6]. This situation is illustrated in Figure 11. Without the aid of the working-set model, one might conjecture that processing efficiency approaches unity as the degree of multiprogramming n (number of active processes) increases; but, in fact, processing efficiency exhibits a marked dropoff when the degree of multiprogramming exceeds a certain critical level n_0. The explanation of this behavior is simply that, when $n > n_0$, there is no longer sufficient space in memory to accommodate all active working sets. The second implication with respect to multiprogramming is closely related to the first. As suggested in Figure 12, the cost of wasted (unused) memory decreases with n, and

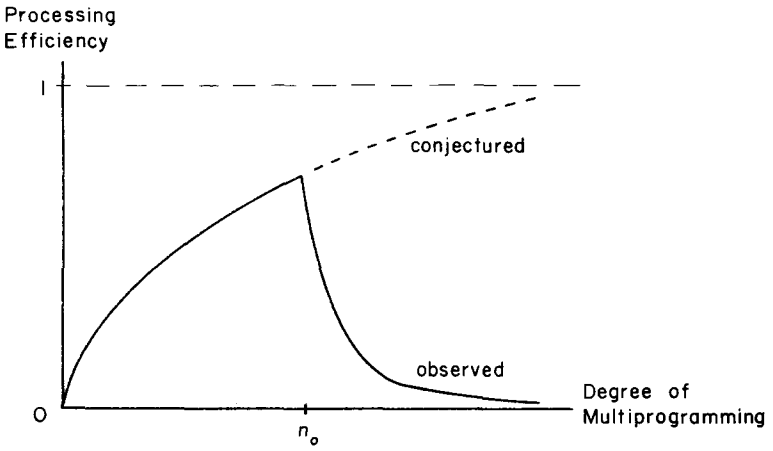


FIG. 11. Thrashing.

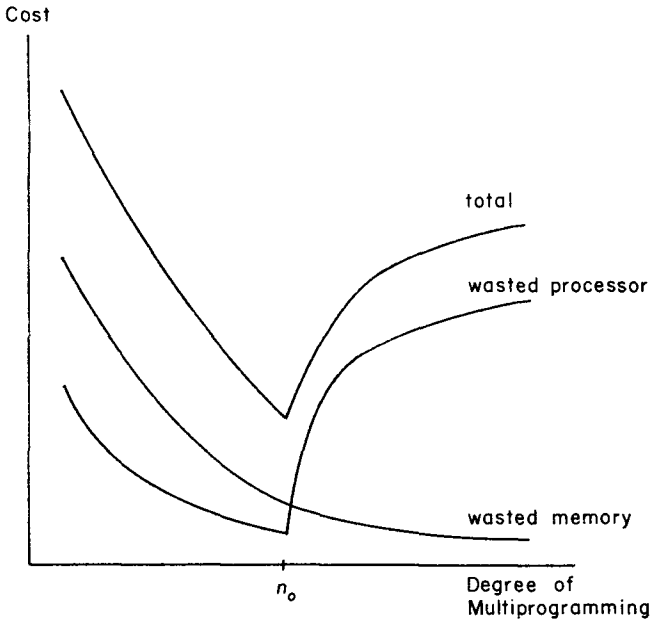


FIG. 12. Cost of multiprogramming.

the cost of wasted processor follows a curve whose shape is similar to the one in Figure 11; at some point in the vicinity of n_0 the total cost is minimum. Thus, it is desirable to load as many working sets as possible into memory; this approximates an optimal degree of multiprogramming.

System Balance

The foregoing considerations illustrate a general phenomenon: overall system performance may suffer if resource usage is not balanced. Stated differently, over- (or under-) commitment of one resource may result in others being wasted. "Balance" can be given a precise definition as follows. At each moment of time, each process has associated with it a vector, its *demand*, whose components reflect the amount of each resource type required by the process at that time. Thus, process i will have a demand $\mathbf{d}_i = (d_{i1}, \cdots, d_{im})$ in which d_{ij} is

the amount of type j resource it requires. The demands $\mathbf{d}_i$ are random variables. The *total demand* $\mathbf{D}$ of the set A of active processes is the vector sum over the demands of all processes i in A:

$$\boldsymbol{D} = \sum_i \boldsymbol{d}_i = \left(\sum_i d_{i1}, \cdots, \sum_i d_{im}\right). \quad (1)$$

The total demand $\mathbf{D}$ is also a random variable. Two vectors are associated with the resources, $\mathbf{p} = (p_1, \cdots, p_m)$ and $\mathbf{c} = (c_1, \cdots, c_m)$, in which p_j is an allowable overflow probability and c_j a capacity of resource type j. The system is "safely committed" if the probability that the total demand $\mathbf{D}$ exceeds the system capacity $\mathbf{c}$ is bounded by the probabilities $\mathbf{p}$; i.e.,

$$\Pr[\mathbf{D} > \mathbf{c}] \leqslant \mathbf{p}. \quad (2)$$

The system is "fully committed" if adding one more ready process to the active set A would violate condition (2). If the system is overcommitted, then for some resource type j,

$$\Pr\left[\sum_{i \text{ in } A} d_{ij} > c_j\right] > p_j. \quad (3)$$

The system is *balanced* with respect to a given degree of multiprogramming (size of A) if the capacities $\mathbf{c}$ have been chosen so that

$$\Pr[\mathbf{D} > \mathbf{c}] = \mathbf{p}. \quad (4)$$

Thus, system balance in the sense of Eq. (4) implies that: 1) the system is fully committed; and 2) resources are fully utilized to the extent allowed by the overflow probabilities. Under these conditions, A can be called the "balance set."

In the case of a system in which the designer is interested in balancing only a single processor and memory, Eq. (4) can be interpreted to mean: 1) the memory is filled with working sets to the extent allowed by the memory overflow probability; 2) processor time is committed so that response time can be guaranteed within some fixed time in the future, depending on processor overflow probability; and 3) the amount of memory is as small as possible. These points have been treated more fully in [D5]. It is interesting to note that Eq. (4) is not difficult to evaluate in practice, since the central limit theorem of probability theory applied to Eq. (1) asserts that the distributions of the components of $\mathbf{D}$ can be approximated by normal distributions.

Eqs. (3) and (4) show that a given system need not be balanced under arbitrary selections of equipment capacities, i.e., the equipment configuration. Only certain equipment configurations are consistent with system balance and given demand distributions of the user community. Other configurations would only give rise to wasted equipment.

Wilkes [W7] has shown that the efficiency of system operation can be improved by implementing the following policy within the balance set. The "life" of a process is the amount of processor time it consumes between interactions with the user or with other processes. The "age" of a process at a given time is the amount of processor time it has been observed to consume since its last interaction. One property of the process-life distribution is that the expected remaining life, given the age, increases monotonically with increasing age to a maximum. Thus, in order to minimize the probability of removing a page of a process from main memory shortly before the remaining life of that process expires (i.e., to give processes whose next interaction is imminent a chance to reach it rapidly), it is desirable to protect the pages of young processes, over those of old ones, from removal. To do this Wilkes proposes the partitioning of processes in memory into a "resident regime" (young processes) and a "swapping regime" (old processes). Similar considerations are made in CTSS [C4].

The foregoing is not the only possible way to approach system balance. Balancing a system has also been called "tuning" and "load leveling." Bernstein and Sharpe have formulated it in terms of regulating job flow so that measured resource-usage functions track desired resource-usage functions [B3]. Wulf has formulated it in a similar way, but with more attention to measuring usage rates and demands [W10]. Wilkes shows how control theory may be applied to achieving stability when the system attempts to balance itself by controlling the number of users accessing the system at any given time [W9].

Choosing a Policy

The notions of system balance given above are quite flexible. In effect, they make a general statement about the total flow in and out of the balance set, and allow the resource allocation policy considerable latitude in choosing which ready processes shall be elements of that flow. In other words, balance can be viewed as a *constraint* within which a resource allocation policy can seek to optimize some measure of user satisfaction. It thus represents a means of compromising between the conflicting goals of efficiency and satisfaction.

Even with the aid of all these abstractions to give him insight, it remains the task of the system designer to determine which considerations and tradeoffs are important in the system at hand. In addition to the various modeling techniques outlined above, various simulation techniques are available to help him evaluate strategies [M1]. Modeling has the advantages of conceptual simplicity and flexibility, and the disadvantages of being sometimes incomprehensible to clients and of resting on oversimplified assumptions that may not reflect accurately the complexities of a given system. Simulation, on the other hand, has the advantages of being comprehensible and allowing realistic detail, the disadvantages of being time consuming and subject to programming and random errors. When optimal strategies are desired, simulation, although valuable for comparing alternatives, has the added limitation of not being useful in locating optima. A combination of simulation and analysis can be used to improve the efficacy of both [G2, K2].

PROTECTION

The protection of information and other objects entrusted to a computer system has been the subject of considerable concern, especially since data banks have become feasible. The term "protection" has been used in various ways, both technical and nontechnical. The subsequent discussion will deal exclusively with technical aspects, i.e., the techniques used within the system to control access by processes to system objects. The nontechnical aspects of protection—preventing systems programmers from reading dumps of sensitive material, checking the identity of users, keeping magistrates from misusing information stored in a data bank, and so on—will not be discussed here; the reader is referred to [H3] for a survey. Neither will the integrity problem—protection against loss of information due to accident, system failure, hardware error, or a user's own mistakes—be discussed here; the reader is referred to [W6] for a description.

The motivations for a protection system are well known. They concern the shielding of processes in the system from error or malice wrought by other processes. No process should be able to read, destroy, modify, or copy another process's information without authorization. No faulty process should be able to affect the correct operation of any other (independent) process. No user should be required to trust a proprietary program, nor should a proprietary program be required to trust its users [D11, L1, L2]. No debugging program should be required to trust the program it debugs.

The following treatment of protection is based on the abstract model for a protection system proposed by Lampson [L1, L2] and refined by Graham [G4]. Although this model is not the only way of studying the problem, it does have the property that many practical protection systems can be deduced from it.

The model for a protection system consists of three parts. First is a set X of *objects*, an object being any entity in the system that must be protected (i.e., to which access must be controlled). Examples of objects include files, segments, pages, terminals, processes, and protection keys. Each object has associated with it an identification number which is unique for all time; and identification numbers, once used, are never assigned again. (MULTICS, for example, uses the time in μsec starting from 1900 as identification numbers for files.) The second part of a protection system model is a set S of *subjects*, a subject being any entity that may access objects. A subject can be regarded as a pair (process, domain), where a "domain"

is a set of constraints within which the process may access certain objects. Examples of domains include supervisor/user states in a machine, the blocks of the multiprogramming partition in the OS/360-MVT or the CDC 6600, and the file directories in MIT's CTSS. Various terms have been used to denote the idea of domain: protection context, protection state, sphere of protection [D11], and ring of protection [G3]. Since subjects must be protected from each other, subjects are also objects, i.e., S is contained in X. The third part of a protection system model is a collection of *access rules* specifying the ways in which a subject may access objects. A method of specifying access rules will be described shortly.

A protection system fulfills two functions: *isolation* and *controlled access*. Isolation refers to constraining each subject so that it can use only those objects to which access has been authorized. Controlled access refers to allowing different subjects to have different types of access to the same object. Isolation is very simple to implement, but controlled access is in fact the culprit complicating most protection systems. To illustrate this point, we shall first consider a protection system that implements isolation only; then we shall study how and why controlled access may be incorporated.

In the simplest case, each subject is a process. Each object is accessible to exactly one subject, i.e., the objects of the system are partitioned according to subjects. Each subject s has a unique identification number i_s. Subjects interact by sending messages. Each message consists of a string of data to which the system prefixes the identification number of the sender; all messages directed to a given subject are collected in a message buffer associated with that subject. When subject s wishes to transmit a message α to subject s', it calls on the system by an operation

send message (α, s');

and the system then deposits the pair (i_s, α) in the message buffer of s'. As will be seen, the principle that guarantees the correct operation of protection is that *the identification number of a sender subject cannot be forged*. In other words, even though the identification numbers may be known to everyone, there is no way for s to send α to s' that can alter the fact that message (i_s, α) is placed in the message buffer of s'.

This very simple protection system is complete from the viewpoint of isolation. Protected calling of subroutines (protected entry points [D11]) is implemented in the obvious way: if s wishes to "call" s', s sends s' a message consisting of the parameter list α and enters a state in which it waits for a reply message (return) from s'; when s' finds message (i_s, α) in its message buffer, it performs its function with parameters α, then replies to s with a message containing the results. There is no way that s can "enter" s' at any point other than the one at which s' inspects its message buffer. By a similar argument, subroutine return is protected. The protection implemented here goes beyond guaranteeing that subroutines call and return only at the intended points; it is clearly possible to guarantee that s' will act on behalf of authorized callers only, since s' can choose to ignore (or report to the supervisor) any message (i_s, α) from an unauthorized caller s. Access to an arbitrary object x in this system is also completely protected: a subject s may access freely any x associated with it; but if x is associated with another subject s', s must send a message to s' requesting s' to access x on its behalf. The RC-4000 system operates in precisely this fashion [B5].

The system described above has several limitations that motivate the addition of a controlled-access mechanism to the protection system:

1) it is not possible to stop a runaway process, since there is no way to force a receiver either to examine his message buffer or to do anything else—which, among other things, makes debugging difficult;
2) there is no systematic, efficient way of sharing objects among subjects; and
3) it necessitates the establishment of a possibly elaborate system of conventions to enable receivers to discover the identification numbers of authorized senders.

It is necessary, therefore, to generalize the foregoing system so that some subjects can

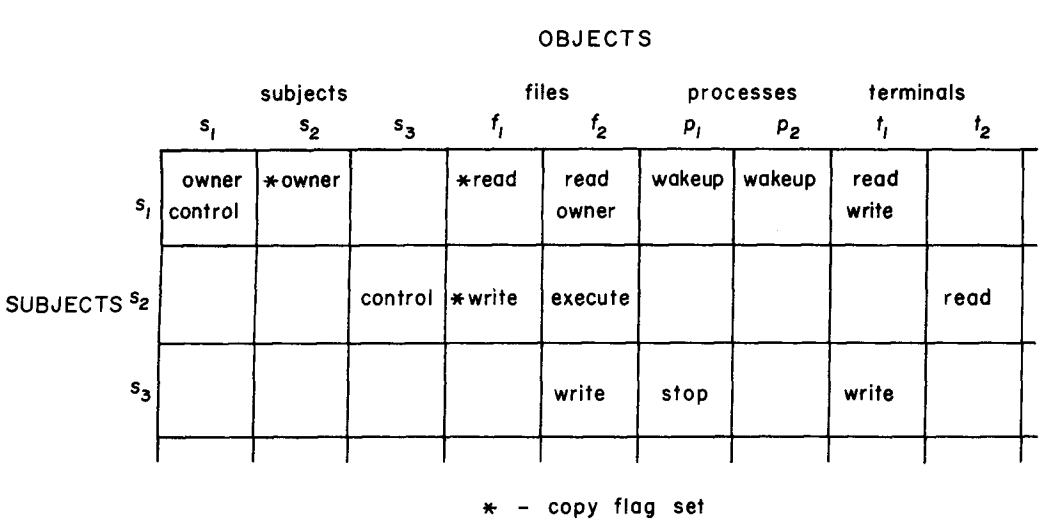


FIG. 13. An access matrix.

exercise control over others, share objects with others, and discover access rights easily. In the generalization, the correctness of the protection system is guaranteed by a generalization of the nonforgery principle stated above: whenever subject s accesses object x, the access is accompanied by the identification number i_s (which cannot be forged), and the protection system permits access only if i_s has been authorized.

The generalized protection system requires an explicit specification of the access rules defining the access that subjects have to objects. (In the simple system, the access rules were implicit in the message transmitting mechanism.) These access rules can be specified in the form of an *access matrix*, A, whose rows correspond to subjects and whose columns correspond to objects. An entry $A[s, x]$ in this matrix contains a list of strings specifying the *access rights* that subject s has to object x. Thus, if string α is in $A[s, x]$, we say that s has access right α to x. For example, in Figure 13,

s_1 is the owner of s_1 and s_2 ;
s_1 may read f_1 ;
s_1 may wakeup p_1 ;
s_2 may execute f_2 ; and
s_3 may stop p_1 .

Each type of object has associated with it a *controller* through which accesses to objects of that type must pass. A subject calls on the appropriate controller when it wishes to access an object. For example:

TYPE OF OBJECT	CONTROLLER
subject	protection system
file	file system
segment or page	memory addressing hardware
process	process manager
terminal	terminal manager

An access would proceed as follows:

1) s initiates access of type α to object x;
2) the system supplies message (i_s, x, α) to the controller of x; and
3) the controller of x asks the protection system whether α is in $A[s, x]$; if so, access is allowed; otherwise, it is denied and a protection violation occurs.

It is important to note that the meaning of a string α in $A[s, x]$ is interpreted by the controller of x during each access to x, not by the protection system. The correctness of this system follows from the system's guarantee that the identification i_s cannot be forged and that the access matrix A cannot be modified except by rules such as those given below.

The three steps above concern the use of the access matrix once its entries have been specified. In addition to these, rules must exist for creating, deleting, and transferring access rights. The rules proposed by Lamp-

son [L2] use the access rights "owner," "control," and "copy flag" (*):

1) *Creating:* s can add any access right to $A[s', x]$ for any s' if s has "owner" access to x. For example, in Figure 13, s_1 can add "control" to $A[s_2, s_2]$ or "read" to $A[s_3, f_2]$.
2) *Deleting:* s can delete any access right from $A[s', x]$ for any x if s has "control" access to s'. For example, s_2 can delete "write" from $A[s_3, f_2]$ or "stop" from $A[s_3, p_1]$.
3) *Transferring:* if $*\alpha$ appears in $A[s, x]$, s can place either $*\alpha$ or α in $A[s', x]$ for any s'. For example, s_1 can place "read" in $A[s_2, f_1]$ or "*owner" in $A[s_3, s_2]$.

The copy flag is required so that one subject can prevent untrustworthy other subjects from maliciously giving away access rights that it has given them.

The three rules given above should be regarded as examples of rules for creating, deleting, and transferring. The exact nature of these rules will depend on the objectives of a given system. For example, according to Graham [G4], there is no need for "owner" and "control" to be distinct. Further, the notion of transferring can be extended to include a "transfer-only" mode according to which the transferred access right disappears from the transferring subject; thus, if the symbol • indicates this mode, then s can place $\bullet\alpha$ or α in $A[s', x]$ whenever $\bullet\alpha$ appears in $A[s, x]$, but, in so doing, $\bullet\alpha$ is deleted from $A[s, x]$. One may wish to limit the number of owners of an object to exactly one; assuming that each object initially has one owner, this condition can be perpetuated by allowing only "• owner" or "owner," but not "*owner."

A practical realization of a protection system can neither represent access rights as strings nor store the access matrix as a two-dimensional array. Access rights are encoded; for example, the entry in $A[s, x]$ might be a binary word whose ith bit is set to 1 if and only if the ith access right is present. The size of the access matrix can be reduced by grouping the objects X into disjoint subsets, called *categories* [G1], using the columns of matrix A to represent these subsets; thus, if α is in $A[x, Y]$ for category Y, then subject s has α access to each and every object in category Y. Storing the matrix A as an array is impractical since it is likely to be sparse. Storing it in the form of a table whose entries are of the form

$$(s, x, A[s, x])$$

is likely to be impractical in systems with many objects: the entire table cannot be kept in fast-access memory, especially since only a few subjects and objects are likely to be active at any given time, nor can it be guaranteed that a search for all x to which a given s has access (or all s having access to a given x) can be performed efficiently.

There are, however, several practical implementations. One of these stores matrix A by columns; i.e., each object x has associated with it a list, called an *access control list*, with entries of the form $(s, A[s, x])$. When a subject attempts to access x, the controller of x can consult the access control list of x to discover $A[s, x]$. This is precisely the method used to control access to files in CTSS and to segments in MULTICS [B2, C8, D2]. A second method stores the matrix A by rows; i.e., each subject s has associated with it a list with entries of the form $(x, A[s, x])$, each such entry being called a *capability*, and the list a *capability list*. When a subject s attempts to access x, the controller of x can consult the capability list of s to discover $A[s, x]$. This is precisely the method proposed by Dennis and Van Horn [D11], and implementations of it are discussed in [L1, W6].

Of the two preceding implementations, the capability list appears to be more efficient than the access control list. In either case, the identification number of the current subject can be stored in a protected processor register, so that attaching it to any access is trivial. If the list to be consulted by an object controller is in a protected, read-only array in fast-access memory, the verification of access rights can be done quite easily. In the case of capability lists, the list has to be in fast-access memory whenever the subject is active. In the case of access control lists, however, the list need not be available in fast-access memory as a particular object

may not be active often enough to warrant keeping its list available at all times.

A third practical implementation is a refinement of the capability-list implementation in which each subject has associated with it a series of lists, one for each possible access right: the list of subject s for access right α, denoted by s_α, can be represented as a binary word whose ith bit is set to 1 if and only if subject s has access right α to the ith object x_i. When s attempts α access to object x_i, the controller checks to see whether or not the ith bit of s_α is 1. A combination of this and the category technique is used on the IDA time-sharing system and has been found quite efficient [G1].

A fourth possible implementation represents a compromise between the access-control-list and the capability-list implementations. Associated with each subject s is an object-key list consisting of pairs (x, k), where x is an object name and k is a "key," and this list is inaccessible to s. Each key k is an encoded representation of a particular access right α, and (x, k) will be placed in the object-key list of s only if the α corresponding to k is in $A[s, x]$. Associated with each object x is a list of "locks," each lock being the encoded representation for some α. When s attempts to access x, the controller of x checks to see whether or not one of the keys of s matches one of the locks on x, and permits access only if a match is found. The system of "storage keys" and "locks" used in the OS/360 is a case of this approach.

In addition to the considerations above, tree-structured naming schemes can be introduced to make the process of locating an object more efficient. In the access-control-list implementation, the objects may be associated with the nodes of a tree and the access control lists attached to these nodes; the CTSS and MULTICS file systems exemplify this idea when the objects are files. In the capability-list implementation, the subjects may be associated with the nodes of a tree and the capability lists attached to these nodes; both the hierarchy of spheres of protection suggested by Dennis and Van Horn [D11] and the hierarchy of processes suggested by Brinch Hansen [B5] exemplify this idea. Implementing a subject-hierarchy has the additional advantage of defining protocol for reporting protection violations: a violation detected in a given subject is reported to the immediately superior subject for action.

How does memory protection, such as was discussed in the third section, "Storage Allocation," fit into this framework? Memory protection is enforced by the addressing and relocation hardware. Two properties of this hardware are exploited by protection systems: 1) only information in the range of an address map is accessible to a process; and 2) protection codes can be associated with each entry in a map and used to check the validity of each and every reference. The requirement that information associated with a given subject be protected can be realized by associating an address space with each subject.* For information object x, then, an entry in the address map f is of the form $(x, f(x), p)$, where p is a protection code. (The code p might contain bits giving read, write, or execute access to object x.) Thus, the address map is a form of a capability list, and the protection rules are enforced by the hardware. Sharing a segment among subjects is handled by putting common entries for that object in the address maps of the various subjects sharing it.

CONCLUSIONS

A few points are worth mentioning in closing. The concepts presented here have demonstrated their utility in the practical task of improving systems design and operation; in fact, most are straightforward generalizations of widely accepted viewpoints. These concepts have also demonstrated their utility in the classroom, easing, as they do, the task of explicating the often difficult principles of operating systems. However, these concepts can be considered, at best,

* Associating a separate address space with each subject is not necessarily efficient. It would, for example, force processes to pass symbolic names of segments or files in messages. These symbolic names would have to be converted to local address-space names in order to take advantage of memory addressing hardware.

only a beginning in the evolution of a "theory" of operating systems, for there remains a multitude of problems and issues for which no viable abstractions have yet been developed. Table II lists the more important of these, together with references to the literature that concerns them.

The importance of evolvable systems—ones that can expand or change indefinitely—appears to be widely preached, though rarely implemented. MIT's CTSS, an extensible system, is an excellent example of a system that evolved into a form very different (and considerably more useful) from that originally implemented. However, for a system to be evolvable, it is not sufficient that it be extensible; it requires a community atmosphere conducive to encouraging users to develop or improve on system services. In other words, evolution will not in fact occur if the users themselves neither gain from it nor contribute to it. A file system, or other program-sharing facility, is an essential catalyst in the evolutionary process.

Although the computing era is still young, it has evolved so rapidly that we have sometimes come to regard as obsolete any idea published more than five years ago. One is frequently struck by the similarities between our current problems and those faced by our predecessors. Many new problems are merely abstractions of old ones. The two most obvious examples are the systems programming language problem and the virtual storage allocation problem. Thus, the importance of being familiar with some of the old ideas—so that we can avoid retracing already-trodden paths—cannot be overstated. (Of course, not every current problem is an abstraction of an old one; the parallel process control problem and the protection problem, for example, are new.)

Two factors that have pervaded and flavored approaches to software design since the mid-1950s are the existence of magnetic core main memory and moving-medium auxiliary memory (e.g., drums, tapes). Three properties of memory systems using these devices have been particularly influential: 1) the time to access a word in main memory is of an order of magnitude slower than logic speeds; 2) the locations of core memory are organized in a linear array; and 3) the speed ratio between auxiliary and main memory is on the order of 10,000 or more. The introduction of new memory technologies (of which the cache store is the first step) is going to remove some or all of these assumptions, perhaps precipitously, and is going to require an extensive re-evaluation of all aspects of computer architecture depending on them. System designers will require abstractions of the kinds presented here, both for assessing the effects of the change in technology on existing designs and for developing efficient implementations of the same abstractions in the new technology.

TABLE II. LITERATURE ON PROBLEMS REQUIRING ABSTRACTIONS

Issue	*References*
Administration	N1, P1, S5, S6, S7, W6, W8
Compatibility	C3, C6
Complexity	C3, D14, H5, S1
Data communications, networks and utilities	A3, A4, D8, P1
Design techniques and correctness	B5, D9, D14
Evolvability	D9, D10
Generality	C3, D9
Performance evaluation	A7, A9, W10
Reliability and recovery	W6
System objectives	A8, B5, C4, C5, C7, D9, D10, I1, P1, W2, W6
Transportability of software	P2, W1

ACKNOWLEDGMENTS

I have profited from discussions with many individuals while preparing this paper. I should particularly like to thank Carol Shanesy (New York City Rand Institute); Dan Berry (Brown University); Edward G. Coffman, Jr. (Penn State University); Jack B. Dennis (MIT); Robert M. Keller (Princeton University); Butler W. Lampson (Xerox Palo Alto Research Center); Peter Wegner (Brown University); and Maurice V. Wilkes (Cambridge University). Most of all, however, I am indebted to the referees.

ANNOTATED BIBLIOGRAPHY

A1. ABATE, J.; AND H. DUBNER. "Optimizing the performance of a drum-like storage." *IEEE Trans. Computers* C-18, 11 (Nov. 1969), 992–997.

A description and analysis of the shortest-access-time-first queueing discipline, which minimizes the accumulated latency time of requests for rotating-medium auxiliary stores.

A2. ACM. "Storage Allocation Symposium" (Princeton, N.J., June 1961). *Comm. ACM* 4, 10 (Oct. 1961).

These papers argue the cases and techniques for static and dynamic storage allocation. They provide a picture of the state of considered thought on storage allocation as of 1961.

A3. ACM. "Proc. Symposium on Operating System Principles" (Gatlinburg, Tenn., Oct. 1967). Some of the papers appear in *Comm. ACM* 11, 5 (May 1968).

These papers provide, collectively, a summary of the ideas through the mid 1960s. The session titles are: Virtual Memory, Memory Management, Extended Core Memory Systems, Philosophies of Process Control, System Theory and Design, and Computer Networks and Communications.

A4. ACM. *Proc. Symposium on Problems in the Optimization of Data Communications Systems* (Pine Mt., Ga., Oct. 1969).

These papers provide a summary of ideas on computer data communications and networks as of 1969. The session titles are: Advanced Networks; Advances in Data Communications Technology; Preprocessors for Data Communications; Legal, Political, and Privacy Considerations; Graphic Displays and Terminals; Systems Optimization; Human Factors; and Software Implications.

A5. ACM. *Proc. 2nd Symposium on Operating Systems Principles.* (Princeton Univ., Oct. 1969).

These papers provide, collectively, a summary of ideas through the late 1960s. The session titles include: General Principles of Operating Systems Design; Virtual Memory Implementation; Process Management and Communications; Systems and Techniques; and Instrumentation and Measurement.

A6. ACM. *Record of the Project MAC Conf. on Concurrent Systems and Parallel Computation* (Wood's Hole, Mass., June 1970).

These papers deal with various aspects of parallel computation, such as Petri Net theory, program schemata, and speed-independent computation. A complete bibliography of pertinent literature is included.

A7. ACM. *Proc. Symposium on System Performance Evaluation* (Harvard Univ., Cambridge, Mass., April 1971).

These papers deal with various aspects of the performance evaluation problem. The session titles are: Instrumentation, Queueing Theoretic Models, Simulation Models, Measurement and Performance Evaluation, and Mathematical Models.

A8. ALEXANDER, M. T. "Time sharing supervisor programs (notes)." Univ. Michigan Computer Center, May 1969.

A comparison of four systems: the Michigan Time Sharing System (MTS), MULTICS, CP/67, and TSS/360.

A9. ARDEN, B. W.; AND D. BOETTNER. "Measurement and performance of a multiprogramming system." In *Proc. 2nd Symposium on Operating Systems Principles* [A5], 130–146.

A description of simple but useful performance evaluation techniques used on the Michigan Time Sharing System.

B1. BELADY, L. A. "A study of replacement algorithms for virtual storage computers." *IBM Systems J.* 5, 2 (1966), 78–101.

A detailed empirical study covering properties of program behavior manifested through demand paging algorithms. Various algorithms are compared for different memory and page sizes. An optimal algorithm, MIN, is proposed.

B2. BENSOUSSAN, A.; C. T. CLINGEN; AND R. C. DALEY. "The MULTICS virtual memory." In *Proc. 2nd Symposium on Operating Systems Principles* [A5], 30–42.

A description of the motivations for, and design of, the segmented address space and directory structure used in MULTICS.

B3. BERNSTEIN, A. J.; AND J. SHARPE. "A policy driven scheduler for a time sharing system." *Comm. ACM* 14, 2 (Feb. 1971), 74–78.

A proposal for resource allocation policy that attempts to allocate resources so that a measure of observed progress "tracks" a measure of desired progress.

B4. BRINCH HANSEN, P. (ED.). *RC-4000 software multiprogramming system.* A/S Regnecentralen, Falkoner Alle 1, Copenhagen F, Denmark, April 1969.

The description of the RC-4000 system: its philosophy, organization, and characteristics.

B5. BRINCH HANSEN, P. "The nucleus of a multiprogramming system." *Comm. ACM* 13, 4 (April 1970), 238–241, 250.

A specification of the bare minimum requirements of a multiprogramming system. It is related to the THE system [D14] in its orientation toward providing an efficient environment for executing parallel processes. See also [B4].

B6. BUZEN, J. "Analysis of system bottlenecks using a queueing network model." In *Proc. Symposium on System Performance Evaluation* [A7], 82–103.

An analysis of a cyclic queue network for a computer system with a single central processor (round-robin scheduling) and a collection of peripheral processors. Some results concerning optimal job flows are given.

C1. COFFMAN, E. G., JR.; M. ELPHICK; AND A. SHOSHANI. "System deadlocks." *Computing Surveys* 3, 1 (June 1971), 67–78.

A survey of the methods for preventing, detecting, and avoiding deadlocks. Includes

summaries of the important analytical methods for treating deadlocks.

C2. COFFMAN, E. G., JR.; AND L. KLEINROCK. "Computer scheduling methods and their countermeasures." In *Proc. AFIPS 1968 SJCC*, Vol. 32, AFIPS Press, Montvale, N.J., 11–21.

A review of the basic time-sharing scheduling models and results, and of ways to gain high priority for service.

C3. CORBATÓ, F. J. "PL/I as a tool for systems programming." *Datamation* **15**, 5 (May 1969), 68–76.

Argues the case for a high-level systems programming language, and recounts the MULTICS experience with PL/I.

C4. CORBATÓ, F. J.; M. MERWIN-DAGGET; AND R. C. DALEY. "An experimental time sharing system." In *Proc. AFIPS 1962 SJCC*, Vol. 21, Spartan Books, New York, 335–344. Also in [R2].

Overview of the CTSS (Compatible Time Sharing System) at MIT, including the proposal for the multilevel feedback queue.

C5. CORBATÓ, F. J.; AND V. A. VYSSOTSKY. "Introduction and overview of the MULTICS system." In *Proc. AFIPS 1965 FJCC*, Vol. 27, Pt. 1, Spartan Books, New York, 185–196. Also in [R2].

A description of the hardware and software facilities for the MULTICS (MULTiplexed Information and Computing Service) system at MIT.

C6. CREECH, B. A. "Implementation of operating systems." In *IEEE 1970 Internatl. Convention Digest*, IEEE Publ. 70-C-15, 118–119.

Stresses the importance of a programming language for systems programming, and reports experience with the Burroughs B6500.

C7. CRITCHLOW, A. J. "Generalized multiprogramming and multiprocessing systems." In *Proc. AFIPS 1963 FJCC*, Vol. 24, Spartan Books, New York, 107–126.

A review tracing the evolution of multiprogramming and multiprocessing techniques and concepts through the early 1960s.

C8. CRISMAN, P. A. (ED.). *The compatible time sharing system: a programmer's guide*. MIT Press, Cambridge, Mass., 1965.

The complete description and specification of CTSS.

D1. DALEY, R. C.; AND J. B. DENNIS. "Virtual memory, processes, and sharing in MULTICS." *Comm. ACM* **11**, 5 (May 1968), 306–312. Also in [A3].

A description of the hardware and software facilities of the GE-645 processor which are used to implement virtual memory, segment linkages, and sharing in MULTICS.

D2. DALEY, R. C.; AND P. G. NEUMANN. "A general-purpose file system for secondary storage." In *Proc. AFIPS 1965 FJCC*, Vol. 27, Pt. 1, Spartan Books, New York, 213–229.

Describes a file directory hierarchy structure, its use and conventions, and implementation problems.

D3. DENNING, P. J. "The working set model for program behavior." *Comm. ACM* **11**, 5 (May 1968), 323–333. Also in [A3].

A machine-independent model for program behavior is proposed and studied.

D4. DENNING, P. J. "Thrashing: its causes and prevention." In *Proc. AFIPS 1968 FJCC*, Vol. 33, Pt. 1, AFIPS Press, Montvale, N.J., 915–922.

A definition of, and explanation for, thrashing (performance collapse due to overcommitment of main memory) is offered, using the working set model [D3] as a conceptual aid. Some ways of preventing thrashing are discussed.

D5. DENNING, P. J. "Equipment configuration in balanced computer systems." *IEEE Trans. Computers* **C-18**, 11 (Nov. 1969), 1008–1012.

Some aspects of designing systems with understandable behavior are defined and applied to the equipment configuration problem.

D6. DENNING, P. J. "Virtual memory." *Computing Surveys* **2**, 3 (Sept. 1970), 153–189.

A survey and tutorial covering the definitions, implementations, policies, and theory of automatic storage allocation.

D7. DENNIS, J. B. "Segmentation and the design of multiprogrammed computer systems." *J. ACM* **12**, 4 (Oct. 1965), 589–602. Also in [R2].

The segmented name space and addressing mechanism are proposed and developed. Comparisons with other name space structures are made.

D8. DENNIS, J. B. "A position paper on computing and communications." *Comm. ACM* **11**, 5 (May 1968), 370–377. Also in [A3].

The problems that face computer networks are outlined and some solutions proposed.

D9. DENNIS, J. B. "Programming generality, parallelism, and computer architecture." In *Proc. IFIP Cong. 1968*, Vol. 1, North-Holland Publ. Co., Amsterdam, 484–492. (Also in MIT Project MAC Computation Structures Group Memo No. 32.)

Defines "programming generality"—the ability to construct programs from collections of modules whose internal operations are unknown—and explores its consequences in system design.

D10. DENNIS, J. B. "Future trends in time sharing systems." MIT Project MAC Computation Structures Group Memo No. 36-1, June 1969.

Classifies time-sharing systems and emphasizes the importance of programming generality in achieving the "information utility" and in solving the "software problem."

D11. DENNIS, J. B.; AND E. C. VAN HORN. "Programming semantics for multiprogrammed computations." *Comm. ACM* **9**, 3 (March 1966), 143–155.

The system requirements for implementing segmentation, protection, sharing, parallelism, and multiprogramming are described in terms of "meta-instructions." The concept of "capability" is introduced.

D12. DIJKSTRA, E. W. "Solution of a problem in concurrent programming control." *Comm. ACM* **8,** 9 (Sept. 1965), 569.

This is the first published solution to the mutual exclusion problem, programmed in standard ALGOL without **lock** and **unlock** machine instructions being available.

D13. DIJKSTRA, E. W. "Cooperating sequential processes." In *Programming languages*, F. Genuys (Ed.), Academic Press, New York, 1968, 43–112.

The first study of parallel processes and concurrent programming, including a well-illustrated and complete development of the minimum requirements on the coordination and synchronization control primitives.

D14. DIJKSTRA, E. W. "The structure of THE-multiprogramming system." *Comm. ACM* **11,** 5 (May 1968), 341–346. Also in [A3].

An overview of a process-oriented, multilevel, highly organized operating system designed so that its correctness can be established a priori. Includes an appendix on the synchronizing primitives *P* and *V*. (THE—Technische Hochschule Eindhoven)

F1. FUCHS, E.; AND P. E. JACKSON. "Estimates of random variables for certain computer communications traffic models." *Comm. ACM* **13,** 12 (Dec. 1970), 752–757.

An empirical study of interarrival and service distributions frequently encountered in computer systems. The exponential and gamma distributions are found to be useful in many cases of practical interest.

G1. GAINES, R. S. "An operating system based on the concept of a supervisory computer." In *Proc. Third Symposium on Operating Systems Principles*, to appear in *Comm. ACM* **15,** 3 (March 1972).

A description of the structure of the operating system on the CDC 6600 computer at IDA (Institute for Defense Analyses), Princeton, N.J. The design is related to that of the RC-4000 [B4, B5].

G2. GAVER, D. P. "Statistical methods for improving simulation efficiency." Carnegie-Mellon Univ., Pittsburgh, Pa., Management Science Research Group Report No. 169, Aug. 1969.

Discusses a number of methods for shortening the computation time of simulations by improving the rate of convergence.

G3. GRAHAM, R. M. "Protection in an information processing utility." *Comm. ACM* **11,** 5 (May 1968), 365–369. Also in [A3].

A description of the "ring" protection structure in MULTICS.

G4. GRAHAM, G. S. "Protection structures in operating systems." Master of Science Thesis, Univ. Toronto, Toronto, Ont., Canada, Aug. 1971.

A detailed investigation of Lampson's metatheory [L2] and its system implications, together with its application to Project SUE at the University of Toronto.

H1. HABERMANN, A. N. "Prevention of system deadlock." *Comm. ACM* **12,** 7 (July 1969), 373–377.

Definition of the deadlock problem, and the "banker's algorithm" method of determining safe allocation sequences.

H2. HAVENDER, J. W. "Avoiding deadlock in multitasking systems." *IBM Systems J.* **7,** 2 (1968), 74–84.

Treats some of the deadlock problems encountered in OS/360, and proposes the "ordered resource usage" method of preventing deadlock.

H3. HOFFMAN, L. J. "Computers and privacy: a survey." *Computing Surveys* **1,** 2 (June 1969), 85–103.

An overview of the philosophical, legal, and social implications of the privacy problem, with emphasis on technical solutions.

H4. HOLT, R. C. "Deadlock in computer systems." PhD Thesis, Rep. TR-71-91, Cornell Univ., Dept. Computer Science, Ithaca, N.Y., 1971.

An exposition of the deadlock problem for systems consisting of both reusable and consumable resources. The emphasis is on graphical methods of description and on efficient algorithms for detecting and preventing deadlocks.

H5. HORNING, J. J.; AND B. RANDELL. "Structuring complex processes." IBM Thomas J. Watson Research Center Report RC-2459, May 1969.

A formal, state-machine-oriented model for the definition, control, and composition of processes.

I1. IBM. "Operating System 360 concepts and facilities." In *Programming Systems and Languages* [R2], 598–646.

Excerpts from IBM documentation on the structure and operation of OS/360.

I2. IEEE. "Proc. 1969 Computer Group Conference." *IEEE Trans. Computers* **C-18,** 11 (Nov. 1969).

Of interest is the session on "Computer system models and analysis."

I3. IRONS, E. T. "Experience with an extensible language." *Comm. ACM* **13,** 1 (Jan. 1970), 31–40.

Another example of achievable success in designing a time-sharing operating system (for the CDC 6600) using an extensible, high-level language.

J1. JOHNSTON, J. B. "The contour model of block structured processes." In *Proc. Symposium on Data Structures in Programming Languages* (Gainesville, Fla., Feb. 1971), J. T. Tou & P. Wegner (Eds.), special issue of *SIGPLAN Notices*, ACM, New York, Feb. 1971.

The contour model is primarily a pictorial way of describing the properties, operation, and implementation of processes generated by programs written in block structured programming languages. A "contour" is the boundary of a local environment of an activated block in the diagrams.

K1. KILBURN, T.; D. B. G. EDWARDS; M. J. LANIGAN; AND F. H. SUMNER. "One-level storage system." *IRE Trans.* **EC-11** (April 1962), 223–238.

The first detailed description of the paging mechanism on the Atlas Computer, including the loop-detecting "learning" algorithm for page replacement.

K2. KIMBLETON, S.; AND C. A. MOORE. "Probabilistic framework for system performance evaluation." In *Proc. Symposium on System Performance Evaluation* [A7], 337–361.

A process is modeled as cycling through the states "blocked," "ready," and "executing." The model is analyzed and shown to predict certain properties of the Michigan Time Sharing System.

K3. KLEINROCK, L. "A continuum of time sharing scheduling algorithms." In *Proc. AFIPS 1970 SJCC*, Vol. 36, AFIPS Press, Montvale, N.J., 453–458.

A piecewise linear priority function is associated with each job, and the scheduler runs the job or jobs of highest priority. A job enters the system with zero priority, it gains priority while waiting according to a "waiting slope," and it gains priority while running according to a "serving slope." A two-dimensional plane displays the possible combinations of the two slopes and the corresponding scheduling algorithms.

L1. LAMPSON, B. W. "Dynamic protection structures." In *Proc. AFIPS 1969 FJCC*, Vol. 35, AFIPS Press, Montvale, N.J., 27–38.

An exposition of the programming implications and power of a system of protection based on the capability idea [D11] with hardware similar to that discussed in [W6].

L2. LAMPSON, B. W. "Protection." In *Proc. 5th Annual Princeton Conf.*, Princeton Univ., March 1971.

A generalized model for protection systems, incorporating an "access matrix" and nonforgeable "domain identifiers," is proposed. Existing protection systems are shown to be special cases. Properties of "correct" protection systems are treated.

L3. LIPTAY, J. S. "Structural aspects of the System/360 model 85: the cache." *IBM Systems J.* **7**, 1 (1968), 15–21.

A description of the organization and behavior of the cache store on the IBM 360/85.

M1. MACDOUGALL, M. H. "Computer system simulation: an introduction." *Computing Surveys* **2**, 3 (Sept. 1970), 191–209.

The reader is introduced to the construction and operation of event-driven simulation techniques by a running example of a multiprogramming system.

M2. MATTSON, R. L.; J. GECSEI; D. R. SLUTZ; AND I. L. TRAIGER. "Evaluation techniques for storage hierarchies." *IBM Systems J.* **9**, 2 (1970), 78–117.

The concept of "stack algorithm" as a model for paging algorithms is introduced and its properties studied. Efficient algorithms are derived for determining the "success function" versus memory size for a given paging algorithm. Priority, optimal, and random-replacement algorithms are shown to be stack algorithms.

M3. MCCARTHY, J.; F. J. CORBATÓ; AND M. MERWIN-DAGGET. "The linking segment subprogram language and linking loader." *Comm. ACM* **6**, 7 (July 1963), 391–395. Also in [R2].

Describes a mechanism for achieving program modularity.

M4. MCKINNEY, J. M. "A survey of analytical time-sharing models." *Computing Surveys* **1**, 2 (June 1969), 105–116.

The important queueing theory models for time-sharing scheduling are defined, and the analytic results stated and discussed.

N1. NIELSEN, N. R. "The allocation of computer resources—is pricing the answer?" *Comm. ACM* **13**, 8 (Aug. 1970), 467–474.

The philosophy, common conceptions, and common misconceptions of pricing structures for computer systems are discussed.

P1. PARKHILL, D. *The challenge of the computer utility*. Addison-Wesley Publ. Co., Reading, Mass., 1966.

A good description of the issues, philosophy, methods, hopes, and fears that existed during the early 1960s, when enthusiasm for the computer utility ran high.

P2. POOLE, P. C.; AND W. M. WAITE. "Machine independent software." In *Proc. 2nd Symposium on Operating Systems Principles* [A5], 19–24.

Describes the requirements and objectives for realizing transportable software. See also [W1].

R1. RANDELL, B.; AND C. J. KUEHNER. "Dynamic storage allocation systems." *Comm. ACM* **11**, 5 (May 1968), 297–306. Also in [A3].

Various automatic storage allocation systems and hardware addressing mechanisms are classified, compared, and defined.

R2. ROSEN, S. (ED.). *Programming systems and languages*. McGraw-Hill, New York, 1967.

A collection of "classic" papers on programming language design, definition, assemblers, compilers, and operating systems to 1966.

R3. ROSEN, S. "Electronic computers: a historical survey." *Computing Surveys* **1**, 1 (March 1969), 7–36.

An overview of electronic computing machines through the mid 1960s. The emphasis is on the properties of the various machines and the forces that led to their development.

R4. ROSIN, R. F. "Supervisory and monitor systems." *Computing Surveys* **1**, 1 (March 1969), 37–54.

A historical overview of second and third generation operating systems. The emphasis is on IBM approaches.

S1. SALTZER, J. H. "Traffic control in a multiplexed computer system." MIT Project MAC Report MAC-TR-30, 1966.

A description of the operating principles of the MULTICS interprocess communication facility.

S2. SAMMET, J. E. *Programming languages.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1969.

Encyclopedic coverage, comparing and contrasting over 100 programming languages.

S3. SAYRE, D. "Is automatic folding of programs efficient enough to displace manual?" *Comm. ACM* **12,** 12 (Dec. 1969), 656–660.

Potent empirical evidence supporting the assertion that automatic storage allocation can be at least as cost-effective as manual storage allocation.

S4. SCHROEDER, M. D. "Performance of the GE-645 associative memory while MULTICS is in operation." In *Proc. Symposium on System Performance Evaluation* [A7], 227–245.

A description of the associative memory used in the MULTICS segmentation and paging hardware, and empirical data on its performance.

S5. SELWYN, L. L. "Computer resource accounting in a time-sharing environment." In *Proc. AFIPS 1970 SJCC*, Vol. 36, AFIPS Press, Montvale, N.J., 119–129.

A description of the management, accounting, and pricing systems of CTSS-like systems. Should be read in conjunction with [N1, S7, W8].

S6. SHARPE, W. F. *The economics of computers.* Columbia Univ. Press, New York, 1969.

This is a self-contained text covering the economic theory peculiar to computers and to system selection and evaluation.

S7. SUTHERLAND, I. E. "A futures market in computer time." *Comm. ACM* **11,** 6 (June 1968), 449–451.

Describes a supply-and-demand pricing system for computer utilities. Should be read in conjunction with [N1, S5, W8].

T1. TRIMBLE, G., JR. "A time-sharing bibliography." *Computing Reviews* **9,** 5 (May 1968), 291–301.

V1. VAN HORN, E. C. "Computer design for asynchronously reproducible multiprocessing." MIT Project MAC Report MAC-TR-34, 1966.

One of the first doctoral dissertations addressing the problems of determinate computation by parallel processes.

W1. WAITE, W. M. "The mobile programming system: STAGE2." *Comm. ACM* **13,** 7 (July 1970), 415–421.

Describes a bootstrap sequence technique for implementing transportable software.

W2. Wegner, P. (Ed.). *Introduction to systems programming.* Academic Press, New York, 1964.

A collection of some "classic" papers on operating systems principles and design during the early 1960s.

W3. WEGNER, P. *Programming languages, information structures, and machine organization.* McGraw-Hill, New York, 1968.

Discusses the general aspects of third generation machine and software organization; assemblers and macro-generators; ALGOL runtime environment implementation; PL/I and simulation languages; and Lambda Calculus machines.

W4. WILKES, M. V. "Slave memories and dynamic storage allocation." *IEEE Trans. Computers* **C-14** (1965), 270–271.

Proposes an automatic multilevel memory system, the predecessor of the "cache store" [L3].

W5. WILKES, M. V. "Computers then and now." *J. ACM* **15,** 1 (Jan. 1968), 1–7.

A thought-provoking piece containing historical perspective and evaluation of trends, written as a Turing Lecture.

W6. WILKES, M. V. *Time sharing computer systems.* American Elsevier Publ. Co., New York, 1968.

A masterful 96-page overview of the properties and problems of CTSS-like time-sharing systems.

W7. WILKES, M. V. "A model for core space allocation in a time sharing system." In *Proc. AFIPS 1969 SJCC*, Vol. 34, AFIPS Press, Montvale, N.J., 265–271.

Uses experience with time-sharing systems to develop a minimal relationship between processor and core allocation policies.

W8. WILKES, M. V.; AND D. F. HARTLEY. "The management system—a new species of software?" *Datamation* **15,** 9 (Sept. 1969), 73–75.

Outlines the motivation and requirements for a management system as illustrated by the Cambridge University Multiple Access System. Should be read in conjunction with [N1, S5, S7].

W9. WILKES, M. V. "Automatic load adjustment in time sharing systems." In *Proc. Symposium on System Performance Evaluation* [A7], 308–320.

A model of the load-leveling mechanism used on CTSS is presented and analyzed using some techniques from control theory. The emphasis is on preventing instability in the system whose load is being controlled.

W10. WULF, W. A. "Performance monitors for multiprogramming systems." In *Proc. 2nd Symposium on Operating Systems Principles* [A5], 175–181.

A proposal for a general method of performance monitoring and control of resource usage rates.