A New Remote-Accessed Man-Machine System

Reprints of the papers on the MULTICS system presented at the Fall Joint Computer Conference, Las Vegas, Nevada, November 30, 1965.

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01). Reproduction of this report, in whole or in part, is permitted for any purpose of the United States Government.

CONTENTS

SESSION 6: A NEW REMOTE ACCESSED MAN-MACHINE SYSTEM

Introduction and Overview of the Multics System	F. J. Corbató V. A. Vyssotsky	185
System Design of a Computer for Time-Sharing Applications	E. L. Glaser J. F. Couleur G. A. Oliver	197
Structure of the Multics Supervisor	V. A. Vyssotsky F. J. Corbató R. M. Graham	203
A General-Purpose File System for Secondary Storage	R. C. Daley P. G. Neumann	213
Communications and Input-Output Switching in a Multiplex Computing System	J. F. Ossanna L. Mikus S. D. Dunten	231
Some Thoughts About the Social Implications of Accessible Computing	E. E. David, Jr. R. M. Fano	243

INTRODUCTION AND OVERVIEW OF THE MULTICS SYSTEM*

F. J. Corbató
Massachusetts Institute of Technology
Cambridge, Massachusetts
and
V. A. Vyssotsky
Bell Telephone Laboratories, Inc.
Murray Hill, New Jersey

Multics (*Multi*plexed *I*nformation and *C*omputing *S*ervice) is a comprehensive, general-purpose programming system which is being developed as a research project. The initial Multics system will be implemented on the GE 645 computer. One of the overall design goals is to create a computing system which is capable of meeting almost all of the present and near-future requirements of a large computer utility. Such systems must run continuously and reliably 7 days a week, 24 hours a day in a way similar to telephone or power systems, and must be capable of meeting wide service demands: from multiple man-machine interaction to the sequential processing of absentee-user jobs; from the use of the system with dedicated languages and subsystems to the programming of the system itself; and from centralized bulk card, tape, and printer facilities to remotely located terminals. Such information processing and communication systems are believed to be essential for the future growth of computer use in business, in industry, in government and in scientific laboratories as well as stimulating applications which would be otherwise undone.

Because the system must ultimately be comprehensive and able to adapt to unknown future requirements, its framework must be general, and capable of evolving with time. As brought out in the companion papers,[1-5] this need for an evolutionary framework influences and contributes to much of the system design and is a major reason why most of the programming of the system will be done in the PL/I language.[6] Because the PL/I language is largely machine-independent (e.g. data descriptions refer to logical items, not physical words), the system should also be. Specifically, it is hoped that future hardware improvements will not make system and user programs obsolete and that implementation of the entire system on other suitable computers will require only a moderate amount of additional programming.

The present paper attempts to give a detailed dis-

*Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

cussion of the design objectives as they relate to the major areas of the system. Some of the highlights of the subsequent papers are: a virtual memory system for each user involving two-dimensional addressing with segmentation and paging; the dynamic linking of program segment cross-references at execution time to minimize system overhead; the routine use of sharable, recursive, pure procedure programming within the system as the normal mode of operation; the pooled use of multiple processors, memory modules, and input-output controllers; and multiprogramming of all resources and of multiple users. Automatic management of the complex of secondary storage media along with backup, retrieval, and maintenance procedures for the stored information will be provided by a file system. Further, it is expected that most of the software of the system will be almost identical in form to user programs. The system will incorporate automatic page-turning for both user and system programs alike.

INTRODUCTION

As computers have matured during the last two decades from curiosities to calculating machines to information processors, access to them by users has not improved and in the case of most large machines has retrogressed. Principally for economic reasons, batch processing of computer jobs has been developed and is currently practiced by most large computer installations, and the concomitant isolation of the user from elementary cause-and-effect relationships has been either reluctantly endured or rationalized. For several years a solution has been proposed to the access problem.[7-9] This solution, usually called time-sharing, is basically the rapid time-division multiplexing of a central processor unit among the jobs of several users, each of which is on-line at a typewriter-like console. The rapid switching of the processor unit among user programs is, of course, nothing but a particular form of multiprogramming.

It is now abundantly clear that it is possible to create a general-purpose time-shared multiaccess system on many contemporary computers (especially after minor but basic modifications are made). Already two major and extensive systems have been created, one on the IBM 7094[10,11] and one on the Q-32 computer.[12] In addition, there have been numerous smaller scale systems, the most notable being on the DEC PDP-1,[13,14] the IBM 7094,[15] the GE-235,[16] the DEC PDP-6,[17] and the SDS 930,[18] as well as somewhat more limited versions of time-sharing on the RW-400,[19,20] and the CDC G21,[21] the Johnniac,[22] and the IBM 7040.[23] As time goes on, surveys of implemented systems are being made[12,24] and "score cards" are being kept.[25]

The impetus for time-sharing first arose from professional programmers because of their constant frustration in debugging programs at batch processing installations. Thus, the original goal was to time-share computers to allow simultaneous access by several persons while giving to each of them the illusion of having the whole machine at his disposal. However, at Project MAC it has turned out that simultaneous access to the machine, while obviously necessary to the objective, has not been the major ensuing benefit.[26] Rather, it is the availability at one's fingertips of facilities for editing, compiling, debugging, and running in one continuous interactive session that has had the greatest effect on programming. Professional programmers are encouraged to be more imaginative in their work and to investigate new programming techniques and new problem approaches because of the much smaller penalty for failure. But, the most significant effect that the MAC system has had on the MIT community is seen in the achievements of persons for whom computers are tools for other objectives. The availability of the MAC system has not only changed the way problems are attacked, but also important research has been done that would not have been undertaken otherwise. As a consequence the objective of the current and future development of time-sharing should extend way beyond the improvement of computational facilities with respect to traditional computer applications. Rather, it is the on-line use of computers for new purposes and in new fields which should provide the challenge and the motivation to the system designer. In other words, the major goal is to provide suitable tools for what is currently being called machine-aided cognition.

More specifically, the importance of a multiple-access system operated as a computer utility is that it allows a vast enlargement of the scope of computer-based activities, which should in turn stimulate a corresponding enrichment of many areas of our society. Over two years of experience indicates that continuous operation in a utility-like manner,

with flexible remote access, encourages users to view the system as a thinking tool in their daily intellectual work. Mechanistically, the qualitative change from the past results from the drastic improvement in access time and convenience. Subjectively, the change lies in the user's ability to control and affect interactively the course of a process whether it involves numerical computation or manipulation of symbols. Thus, parameter studies are more intelligently guided; new problem-oriented languages and subsystems are developed to exploit the interactive capability; many complex analytical problems, as in magnetohydrodynamics, which have been too cumbersome to be tackled in the past are now being successfully pursued; even more, new, imaginative approaches to basic research have been developed as in the decoding of protein structures. These are examples taken from an academic environment; the effect of a multiple-access system on business and industrial organizations can be expected to be equally dramatic but experience in this area is still very limited. It is with such new applications in mind that the Multics system has been developed. Not that the traditional uses of computers are being disregarded. Rather, these needs are viewed as a subset of the broader more demanding requirements of the former.

To meet the above objectives, issues such as response time, convenience of manipulating data and program files, ease of controlling processes during execution and above all, protection of private files and isolation of independent processes become of critical importance. These issues demand departures from traditional computer systems. While these departures are deemed to be desirable with respect to traditional computer applications, they are essential for rapid man-machine interaction.

SYSTEM REQUIREMENTS

In the early days of computer design, there was the concept of a single program on which a single processor computed for long periods of time with almost no interaction with the outside world. Today such a view is considered incomplete; for the effective boundaries of an information processing system extend beyond the processor, beyond the card reader and printer and even beyond the typing of input and the reading of output. In fact they encompass as well what several hundred persons are trying to accomplish. To better understand the effect of this broadened design scope, it is helpful to examine several phenomena characteristic of large service-oriented computer installations.

First, there are incentives for any organization to have the biggest possible computer system that it can afford. It is usually only on the biggest computers that there are the elaborate programming systems, compilers and features which make a computer "powerful." This comes about partly because it is more difficult to prepare system programs for smaller computers when limited by speed or memory size and partly because the larger systems involve more persons as manufacturers, managers, and users and hence permit more attention to be given to the system programs. Moreover, by combining resources in a single computer system, rather than in several, bulk economies and therefore lower computing costs can be achieved. Finally, as a practical matter, considerations of floor space, management efficiency and operating personnel provide a strong incentive for centralizing computer facilities in a single large installation.

Second, the capacity of a contemporary computer installation, regardless of the sector of applications it serves, must be capable of growing to meet a continuously increasing demand. A doubling of demand every two years is not uncommon.[27] Multiple-access computers promise to accelerate this growth further since they allow a man-machine interaction rate which is faster by at least two orders of magnitude. Present indications are that multiple-access systems for only a few hundred simultaneous users can generate a demand for computation exceeding the capacity of the fastest existing single-processor system. Since the speed of light, the physical sizes of computer components, and the speeds of memories are intrinsic limitations on the speed of any single processor, it is clear that systems with multiple processors and multiple memory units are needed to provide greater capacity. This is not to say that fast processor units are undesirable, but that extreme system complexity to enhance this single parameter among many appears neither wise nor economic.

Third, computers are no longer a luxury used when and if available, but primary working tools in business, government, and research laboratories. The more reliable computers become, the more their availability is depended upon. A system structure including pools of functionally identical units

(processors, memory modules, input/output controllers, etc.) can provide continuous service without significant interruption for equipment maintenance, as well as provide growth capability through the addition of appropriate units.

Fourth, user programs, especially in a time-sharing system, interact frequently with secondary storage devices and terminals. This communication traffic produces a need for multiprogramming to avoid wasting main processor time while an input/output request is being completed. It is important to note that an individual user is ordinarily incapable of doing an adequate job of multiprogramming since his program lacks proper balance, and he probably lacks the necessary dynamic information, ingenuity or patience.

Finally, as noted earlier, the value of a time-sharing system lies not only in providing, in effect, a private computer to a number of people simultaneously, but, above all, in the services that the system places at the fingertips of the users. Moreover, the effectiveness of a system increases as user-developed facilities are shared by other users. This increased effectiveness because of sharing is due not only to the reduced demands for core and secondary memory but also to the cross-fertilization of user ideas. Thus a major goal of the present effort is to provide multiple access to a growing and potentially vast structure of shared data and shared program procedures. In fact, the achievement of multiple access to the computer processors should be viewed as but a necessary subgoal of this broader objective. Thus the primary and secondary memories where programs reside play a central role in the hardware organization and the presence of independent communication paths between memories, processors and terminals is of critical importance.

From the above it can be seen that the system requirements of a computer installation are not for a single program on a single computer, but rather for a large system of many components serving a community of users. Moreover, each user of the system asynchronously initiates jobs of arbitrary and indeterminate duration which subdivide into sequences of processor and input/output tasks. It is out of this seemingly chaotic, random environment that one arrives at a utility-like view. For instead of chaos, one can average over the different user requests to achieve high utilization of all resources. The task of multiprogramming required to do this need only be organized once in a central supervisor program. Each user thus enjoys the benefit of efficiency without having to average the demands of his own particular program.

With the above view of computer use, where tasks start and stop every few milliseconds and where the memory requirements of tasks grow and shrink, it is apparent that one of the major jobs of the supervisor program (i.e., "monitor," "executive," etc.) is the allocation and scheduling of computer resources. The general strategy is clear. Each user's job is subdivided into tasks, usually as the job proceeds, each of which is placed in an appropriate queue (i.e., for a processor or an input/output controller). Processors or input/output controllers are in turn assigned new tasks as they either complete or are removed from old tasks. All processors are treated equivalently in an anonymous pool and are assigned to tasks as needed; in particular, the supervisor does not have a special processor. Further, processors can be added or deleted without significant change in either the user or system programs. Similarly, input/output controllers are directed from queues independently of any particular processor. Again, as with the processors, one can add or delete input/output capacity according to system load without significant reprogramming required.

THE MULTICS SYSTEM

The overall design goal of the Multics system is to create a computing system which is capable of comprehensively meeting almost all of the present and near-future requirements of a large computer service installation. It is not expected that the initial system, although useful, will reach the objective; rather the system will evolve with time in a general framework which permits continual growth to meet unknown future requirements. The use of the PL/I language will allow major system software changes to be developed on a schedule separate from that of hardware changes. Since most organizations can no longer afford to overlap old and new equipment during changes, and since software development is at best difficult to schedule, this relative machine-independence should be a major asset.

It is expected that the Multics system will be published when it is operating substantially and will therefore be available for implementation on any

equipment with suitable characteristics. Such publication is desirable for two reasons: First, the system should withstand public scrutiny and criticism volunteered by interested readers; second, in an age of increasing complexity, it is an obligation to present and future system designers to make the inner operating system as lucid as possible so as to reveal the basic system issues.

The accompanying papers describe in some detail how the Multics system will meet its objectives. However, it is useful, in establishing an overview, to touch on the highlights and especially on the design motivation.

DESIGN FEATURES OF THE HARDWARE

The Multics system objectives required equipment features that were not present in any existing computer. Consequently it was necessary to develop for the Multics system the GE 645 computer. The GE 635 computer was selected for modification to the GE 645 inasmuch as it already satisfied many of the crucial requirements. In particular, it was designed to have multiprocessors, multiple memory modules, and multiple input/output controllers. Thus, the requirements of modular construction for reliability and for ease of growth were amply met. The communication pattern is particularly straightforward since there are no physical paths between the processors and the input/output equipment; rather all communication is done by means of "mailboxes" in the memory modules and by corresponding interrupts. Furthermore, major modules of the system communicate on an asynchronous basis; thus, any single module can be upgraded without any changes to the other modules. This latter property is useful in that one of the ways in which system capacity (and cost) may be regulated is by changing either the speed or number of memory modules. Of course further adjustment of system capacity is possible by varying the number of processor units or the configuration of drum and disk equipment. In any case, one obtains the important simplification that a single supervisor program can operate without substantial change on any configuration of equipment.

Figure 1 illustrates the equipment configuration of a typical Multics system. All central processors (CPU) and Generalized Input/Output Controllers (GIOC) have communication paths with each of the memory modules. When necessary for maintenance or test purposes, the system can be partitioned into two independent systems (although each of the drum, disk and tapes must belong to one of the two systems). The remote terminals can dial either of the two GIOC through the private branch exchange, which is not shown in the figure.

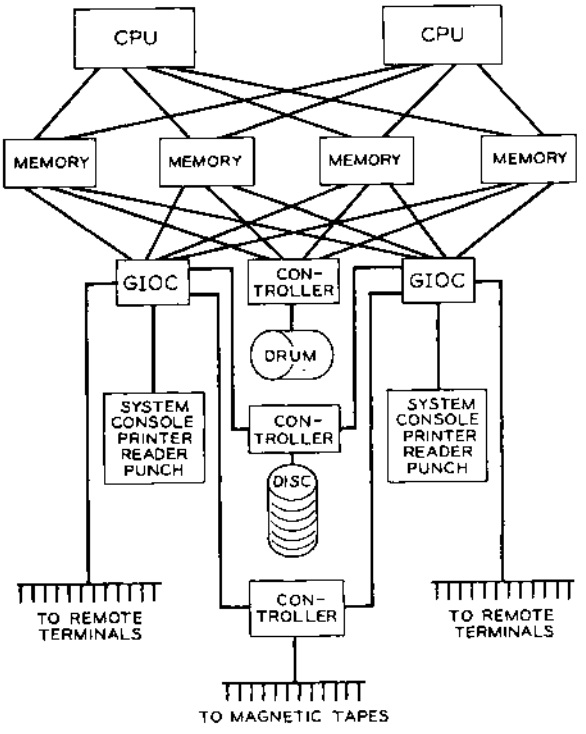

Figure 1. Example of Multics system configuration.

The most novel feature in the GE 645 is in the instruction addressing. A two-dimensional addressing system has been incorporated which allows each user to write programs as though there is a virtual memory system of large size. This system is organized into program segments (i.e., regions) each of which contains an ordered sequence of words with a conventional linear address. These segments, which can vary in length during execution, are paged at the discretion of the supervisor program with either 64- or 1,024-word pages. This dual page size allows the supervisor program to use more effective strategies in the handling of multiple users. Paging, first introduced on the Atlas computer,[28] allows flexible dynamic memory allocation techniques as well as the sensible implementation of a one-level store system. To the user in the Multics system, page addressing is invisible; rather, it is the segments which are explicitly known to him and to which he is able to refer symbolically in his programs. These notions were first suggested by Holt,[29] further developed by Den-

nis,[30,31] Dennis and Glaser,[32] Forgie,[33] and others.[34,35] The value of segmentation and paging has since been widely discussed during the past year and has gained broader acceptance.[36-39] The explicit hardware implementation details of segmentation and paging for the Multics system are discussed in the companion paper by Glaser, Couleur and Oliver.[1]

Because two-dimensional addressing is rather new, it is useful to clarify the reasons for it.

The major reasons for segments are:

1. The user is able to program in a two-dimensional virtual memory system. Thus, any single segment can grow (or shrink) during execution (e.g., in the GE 645, each user may have up to a quarter million segments, each including up to a quarter million words).
2. The user can, by merely specifying a starting point in a segment, operate a program implicitly without prior planning of the segments needed or of the storage requirements. For example, if an error diagnostic segment is unexpectedly called for, it is brought in automatically by the supervisor; it is never brought in unless needed. Similarly, elaborate computations which branch into many different segments in a data-dependent way use segments only as needed.
3. The largest amount of code which must be bound together as a solid block is a single segment. Since binding pieces of code together (sometimes called "loading") is a process similar to assembling or compiling, the advantage of being able to prepare an arbitrarily large program as a series of limited-overhead segment bindings is significant. The saving in overhead is comparable to that in FORTRAN when one uses multiple subprograms instead of a single large combined block of statements. If the combined block is used, not only does the compilation process become particularly cumbersome but the eradication of programming errors in all the different sections requires more compilation time.
4. Program segments appear to be the only reasonable way to permit pure procedures and data bases to be shared among several users simultaneously. Pure procedure programs, by definition, do not modify themselves. Therefore a supervisor program can minimize the core memory requirements of a collection of user programs by supplying only one copy of a jointly used pure procedure. Nearly all of the Multics system as well as most of the user programs will be written in this form. One consequence is that there will be no clearcut demarcation between user programs and system programs; instead the demarcation will depend largely on the responsibility for maintenance.

Pages are a separate feature from segments and have further and distinct advantages.

1. The use of paged memory allows flexible techniques for dynamic storage management without the overhead of moving programs back and forth in the primary memory. This reduced overhead is important in responsive time-shared systems where there is heavy traffic between primary and secondary memories.
2. The mechanism of paging, when properly implemented, allows the operation of incompletely loaded programs; the supervisor need only retain in main memory the more active pages, thus making more effective use of high-speed storage. Whenever a reference to a missing page occurs, the supervisor must interrupt the program, fetch the missing page, and reinitiate the program without loss of information.

A critical feature in the segment and paging hardware is the descriptor bit mechanism which controls the access of processors to the memory. These bits essentially allow hardware "fire-walls" to be established within the programming system which assist the isolation of hardware or software difficulties. Besides controlling the usual properties such as read-only, data-only, etc., one descriptor bit allows a segment to be declared "execute-only." The presence of this bit allows procedures to be transferred to and executed but never read by user programs. This feature will be of interest to commercial service bureaus, and in application areas where privacy of program procedure is essential (e.g., a class-room grading program). Another property of the descriptors is that they allow most of the supervisor modules to be written with

the same descriptors as user programs; most system programs thereby do not have access to privileged instructions, the inadvertent use of which can cause drastic machine misbehavior. This feature is especially pertinent when it is recognized that time-sharing systems are real-time systems with behavior which it is difficult to duplicate or repeat. Consequently, all possible compartments and protection mechanisms that one can have are of value.

For effective operation of the Multics system, a drum with a high transfer rate is needed. The drum provided with the GE 645 meets the requirement and allows convenient and efficient management of a high rate of input/output requests. In particular, requests are organized by the supervisor program into queues in core memory and are fetched from these queues by the drum controller asynchronously of the processors. Because of the queues and because drum record sizes are commensurate with core memory page sizes, it is straightforward to program for continuous input/output transmission without latency delays.

Disk input/output requests are also organized into queues and are fetched from core memory by the generalized input/output controller. This controller is discussed in more detail in the paper by Ossanna et al.[4] Again, because the supervisor is contending with a statistical mix of user and supervisor requests for information to and from disk, it is expected that latency delays between requests will be negligible. Because the transmission capacity to the disk is large, system performance is expected to be unhampered by input/output bottlenecks.

Since the Multics system will be used as an information processor in a wide range of applications, it is important that a readable character set be used. The standard character set will be the recently proposed ASCII code which has 128 codes and includes upper and lower case letters.[40] This set, which contains 95 printing graphics, can be reasonably represented on contemporary input/output consoles. Line printers capable of printing the 95 graphics will be standard equipment.

DESIGN FEATURES OF THE SOFTWARE

An important aspect of the software is the subroutine and linkage conventions which are associated with the use of the segment and paging hardware. The following features are incorporated.

1. Any segment has to know another segment only by symbolic name. Intersegment binding occurs dynamically as needed during program execution. Intersegment binding is automatic (i.e., not explicitly programmed by the user) and the mechanism operates at high efficiency after the first binding occurs.
2. Similarly, a segment is able to reference symbolically a location within another segment. This reference binds dynamically and automatically; after binding occurs the first time, program execution is at full speed.
3. It is straightforward for procedures to be pure procedures, capable of being shared by several users.
4. Similarly, it is straightforward to write recursive procedures (i.e., subroutines capable of calling on themselves either directly or indirectly by a circular chain of calls).
5. The general conventions are such that the call, save, and return macros used to link one independently compiled procedure to another do not depend on whether or not the two procedures are in the same segment.
6. Each user is provided with a private software "stack" for temporary storage within each subroutine. Of course, any user can choose to ignore this storage mechanism, but it is available and does not have to be added as an afterthought by a subsystem designer.

In addition, there is basically only one kind of calling sequence, thus avoiding much confusion. System programming is done with the same facilities, tools, etc., available to the ordinary user, and system programs do not have to be written with special forethought. It is anticipated that the system will be open-ended and will be largely created by the users themselves; many of the useful languages and subsystems will undoubtedly be contributed without solicitation. For this reason supervisor and user programs are constructed with similar form, and processes such as paging do not distinguish between user and supervisor programs. (Of course, a few key pieces of the supervisor are locked in core memory.) Thus there is no intrinsic limit on the size of the supervisor program nor on the complexity or the features which it may have. The avoidance

of a size limitation will be of major value as the system services grow.

It is important to recognize that the average user of the system will see no part of the segmentation and paging complexity described in the paper by Glaser et al. Instead he will see a virtual machine with many system characteristics which are convenient to him for writing either single programs or whole subsystems. As a subsystem writer he must be able to make the computer appear to have any particular form ranging from an airline reservations system, to an inventory control system, from a management gaming machine, to even a "FORTRAN *machine*" if so desired. There are no particular restrictions on the kinds of new systems or languages which can be imbedded.

Further features which should ultimately appear in the system are:

1. the ability to have one process spawn other processes which run asynchronously on several processors (thus improving the real-time response of the overall process);
2. the ability for data bases to be shared among simultaneously operating programs.

In addition the system will include all the major features of the present Project MAC system such as interconsole messages and macro-commands. The latter allow users to concatenate sequences of console-issued commands as short programs thereby forming more elaborate commands which can be used with a single name and parameter call.

Another feature of the system is that it will include batch processing facilities as a subset. In particular, users will start processes which may have n terminals attached, with $n=1$ for individual man-machine interaction, and $n=0$ for running an absentee-user program, the latter case corresponding to batch processing. A user will be able to transform conveniently a process back and forth between the zero and one terminal states. In addition, for the purposes of teaching machines and gaming experiments, it will be possible to attach to a process an arbitrary number of additional terminals.

The supervisor will, of course, do scheduling and charging for the use of resources. Scheduling policies will be similar but more general than those currently in the MAC system; for batch processing, jobs should be scheduled so that a user will be able to obtain a quotation of maximum completion time. The time accounting done by the system will be accurate to a few microseconds. In particular, the system will "fight back" by charging for exactly what equipment is used (or others are prevented from using). In this way, orderly system expansion will be possible since the particular equipment charges which are collected will always allow further acquisition of equipment. In addition the system will incorporate hierarchal control of resource allocations and accounting authorizations. A project manager will be able to give computing budgets to group leaders who in turn will be able to delegate flexibly and straightforwardly sub-budgets to team leaders, etc. An important aspect of this resource allocation and budgeting is the ability of any member of the hierarchy to reallocate flexibly those resources over which he has control. With control of the resource allocation and administrative accounting decentralized, the operation of systems which serve hundreds of persons becomes manageable.

In a similar way, system programming is decentralized. For example, the maintenance of the system might not be entirely under the control of a single group; instead particular translators might be delegated to independent subgroups of system programmers. This isolation and distribution of responsibility is considered mandatory for the growth of large, effective systems. Hierarchal and decentralized accounting and system programming is made possible by a highly organized file system which controls the access rights to the secondary memory of the system and thus to the file copies of the vital procedures and data of the system.

DESIGN CONSIDERATIONS IN THE FILE SYSTEM

The file system is a key part of a time-sharing or multiplexed system. It is a memory system which gives the users and the supervisor alike the illusion of maintaining a private set of segments or files of information for an indefinite period of time. This retention is handled by automatic mechanisms operated by the supervisor and is independent of the complex of secondary storage devices of different capacity and access. A scheme, such as is described in the paper by Daley and Neumann,[3] where all files of information are referred to by symbolic name and not by address, allows changes in the secondary storage complex for reasons either of reliability or capacity. In particular, the user is never responsible

for having to organize the movement of information within the secondary storage complex. Instead the file system has a strategy for arranging for high-speed access to recently used material.

Of considerable concern is the issue of privacy. Experience has shown that privacy and security are sensitive issues in a multi-user system where terminals are anonymously remote. For this reason, each user's files can be arranged to be completely private to him. In addition, a user may arrange to allow others to access his files selectively on a linking basis. The linking mechanism permits control over the degree of access one allows (e.g., a user may wish a file to be read but not written). The file system allows files to be simultaneously read but automatically interlocks file writing.

The file system is designed with the presumption that there will be mishaps, so that an automatic file backup mechanism is provided. The backup procedures must be prepared for contingencies ranging from a dropped bit on a magnetic tape to a fire in the computer room.

Specifically, the following contingencies are provided for:

1. A user may discover that he has accidentally deleted a recent file and may wish to recover it.
2. There may be a specific system mishap which causes a particular file to be no longer readable for some "inexplicable" reason.
3. There may be a total mishap. For example, the disk-memory read heads may irreversibly score the magnetic surfaces so that all disk-stored information is destroyed.

The general backup mechanism is provided by the system rather than the individual user, for the more reliable the system becomes, the more the user is unable to justify the overhead (or bother) of trying to arrange for the unlikely contingency of a mishap. Thus an individual user needs insurance, and, in fact, this is what is provided.

DESIGN CONSIDERATIONS IN THE COMMUNICATION AND INPUT/OUTPUT EQUIPMENT

A design feature of the system is that users can view most input/output devices uniformly. Thus a program can read from either a terminal or a disk file, or output can be sent either to a file or to a punch, a typewriter, or a printer. In particular, the user of the system does not have to rewrite his program to change these assignments from day to day or from use to use. The symmetric use of equipment is, of course, highly desirable and makes for greater simplicity and flexibility.

A typical configuration of the Multics system will contain batch processing input/output devices such as card readers, punches and printers and these normally will be centrally located at the main computing installation. For remote users there will be terminals such as the Model 37 Teletype which uses the revised ASCII code with upper and lower case letters. The Model 37 Teletype also can operate on the TWX network of the Bell System. It will therefore be possible for many of the 60,000 TWX subscribers to be, if authorized, users of a Multics installation. An additional standard terminal for the Multics system will be a modified version of the IBM 1052 console. This unit (and all other terminal devices which do not have the ASCII character set) will have software escape conventions, defined to allow unambiguous input or output of the complete ASCII character set. The escape conventions are general and allow even primitive devices (in a graphic sense) to communicate with the system. The IBM 1052 terminals, which basically use the Selectric typewriter mechanism, are operated with a special typeball, prepared for Project MAC as a compromise subset of the ASCII graphics.

For those users who wish to have remotely located satellite substations capable of punching and reading cards and line printing, there are a variety of options available. Because the design of the General Input/Output Controller is relatively flexible, it is possible to use the GE 115, the Univac 1004, or virtually any other similar subcomputer as a terminal, provided one is prepared to implement the necessary interface program modules within a Multics system. At present none of these terminals are completely satisfactory since the full 128-code revised ASCII character set is not standard and excessive use of the software escape mechanism is required for printing.

In general, the area of remote terminal equipment is considered to be in an early state of development. Equipment innovations are expected, as it becomes evident that systems are capable of supporting their use. Terminals with graphical in-

put/output are highly desirable although at present costly. The initial approach of the Multics system will be such that there will be no standard graphical input/output terminal although several special projects are being attempted. The system viewpoint initially will be that all graphical input/output will be with small, dedicated computers capable of handling the immediate interrupts. These small computers may multiplex a few terminals and in turn appear to be not too demanding to the main system. Thus the main system interrupt load will not become excessive. In a similar way the need for real-time instrumentation such as in monitoring experimental apparatus is expected to be handled initially on a nonstandard basis. The philosophy is the same as with graphical input/output, namely, to employ small, dedicated computers for handling the real-time interrupts so as to draw upon the main system for major processing of information in a more leisurely way.

GENERAL CONSIDERATIONS

It is expected that the ultimate limitation on the exploitation of the Multics system will be the knowledge which the user has of it. As a consequence, documentation of what the system contains is considered to be one of the most important aspects of the system. For this purpose a technique has been developed wherein the main system reference manual is to be maintained on-line in a fashion similar to what is currently being done at Project MAC. This allows any user of the system to obtain a current table of contents with changes listed in reverse chronological order. Thereby he can keep abreast of all system changes. Because the manual text is on-line, one is able to obtain immediate access to the latest changes at any hour or at any terminal. The on-line storage of the text also lets the system documentation group, by using appropriate editing programs, make global revisions whenever necessary. Of course, the distribution of manual revisions will still be handled in the ordinary way in that revised manual sections will be available at document rooms. Furthermore, it should be clear that there is no substitute for a good editor maintaining discipline over the documentation and for intelligent selectivity in the reference material. A documentation technique such as the one given here is believed to be an absolute necessity when users of the system no longer visit a computation center in the course of their daily activities. The user who is 200 miles away from the computer installation should have nearly the same knowledge about the system as the one who is 20 feet away.

Another area of consideration is that of compatibility with batch processing. In the Multics system for the GE 645, it will be possible to use simultaneously, but independently, the GECOS batch-processing system; user jobs operating under GECOS should behave exactly as they do on the GE 625 or GE 635 computers. Effort will be made to allow the GECOS user to change conveniently to the Multics frame of operation but there will be no particular attempt made for compatibility between the two systems of basically different design. A user of the GECOS system may continue to use the GECOS system until he is prepared to make a change to the Multics system at his own place, time, and choosing. This, of course, relieves a manager installing a Multics system of the transient effect of several hundred persons changing their computing habits in one day and thus allows distribution of the normal dissatisfaction that arises under such circumstances.

One of the inevitable questions asked of a multiple-access system is what capacity it will have for simultaneous on-line users. The answer, of course, is highly dependent upon what the users are doing. Clearly, if they are requesting virtually nothing, one can have a nearly infinite number of terminals. Conversely, if one person wishes, for a single problem, system resources which equal the entire computing system, it is conceivable, if the scheduling policy allows it, that there can be only one terminal attached to the system. If one assumes that the service requirements are similar to those which have been experienced at Project MAC, then on the basis of simple scaling of processor and memory speed it is expected that the system will be able to serve simultaneously a few hundred users. But it is hazardous to predict any firm numbers; rather the pertinent parameters in a system of this type will always be the cost-performance figures. Performance, of course, is somewhat subjective, but the issues are not those of memory speed, processor speed or input/output speed. Instead the user should judge a system by the quality and variety of services, the response times, the reliability, the overall ease of understanding the system, and the

performance with respect to the interface of the system which he uses. For example, pertinent questions for a PL/I user to ask are how costly, on the average, the translator is per statement, how easy it is to debug the language, and how efficiently the object code produced by the translator runs. Here, the object code referred to is that for an entire problem and not just for isolated "kernels"; the efficiency refers to the total resource drain required to execute the problem and thereby includes the input/output demands as well.

CONCLUSIONS

The present plans for the Multics system are not unattainable. However, it is presumptuous to think that the initial system can successfully meet all the requirements that have been set. The system will evolve under the influence of the users and their activities for a long time and in directions which are hard to predict at this time. Experience indicates that the availability of on-line terminals drastically changes user habits and these changes in turn suggest changes and additions to the system itself.

It is expected that most of the system additions will come from the users themselves and the system will eventually become the repository of the procedure and data knowledge of the community. The Multics system will undoubtedly also open up large classes of new uses not only in science and engineering but also in other areas such as business and education. Just as introduction of higher-level programming languages, such as FORTRAN, increased by an order of magnitude the number of persons using computers, multiple-access systems operated as a utility will substantially extend the exploitation of information processing systems to the point of having significant social consequences. Such social issues are explored in a companion paper by David and Fano.[5]

REFERENCES

1. E. L. Glaser, J. F. Couleur and G. A. Oliver, "System Design of a Computer for Time-Sharing Applications," this volume.

2. V. A. Vyssotsky, F. J. Corbató and R. M. Graham, "Structure of the Multics Supervisor," this volume.

3. R. C. Daley and P. G. Neumann, "A General Purpose File System for Secondary Storage," this volume.

4. J. F. Ossanna, L. E. Mikus and S. D. Dunten, "Communications and Input/Output Switching in a Multiplex Computing System," this volume.

5. E. E. David, Jr., and R. M. Fano, "Some Thoughts About the Social Implications of Accessible Computing," this volume.

6. "IBM Operating System/360, PL/I: Language Specifications," File No. S360-29, Form C28-6571-1, I.B.M. Corp.

7. C. Strachey, "Time Sharing in Large Fast Computers," *Proceedings of the International Conference on Information Processing, UNESCO,* June 1959, paper B. 2. 19.

8. J. C. R. Licklider, "Man-Computer Symbiosis," *IRE Transactions on Human Factors in Electronics,* vol. HFE-1, no. 1, pp. 4-11 (Mar. 1960).

9. J. McCarthy, "Time-Sharing Computer Systems," *Management and the Computer of the Future* (M. Greenberger, ed.), M.I.T. Press, Cambridge, Mass, 1962, pp. 221-236.

10. F. J. Corbató, M. M. Daggett and R. C. Daley, "An Experimental Time-Sharing System," *Proceedings of the Spring Joint Computer Conference, 21,* Spartan Books, Baltimore, 1962, pp. 335-344.

11. F. J. Corbató et al, *The Compatible Time-Sharing System: A Programmer's Guide,* 1st ed., M.I.T. Press, Cambridge, Mass., 1963.

12. J. Schwartz, A General Purpose Time-Sharing System, *Proceedings of the Spring Joint Computer Conference, 25,* Spartan Books, Washington, D.C., 1964, pp. 397-411.

13. J. B. Dennis, "A Multiuser Computation Facility for Education and Research," *Comm. ACM,* vol. 7, pp. 521-529 (Sept. 1964).

14. S. Boilen et al, "A Time-Sharing Debugging System for a Small Computer," *Proceedings of the Spring Joint Computer Conference, 23,* Spartan Books, Baltimore, 1963, pp. 51-58.

15. H. A. Kinslow, "The Time-Sharing Monitor System," *Proceedings of the Fall Joint Computer Conference, 26,* Spartan Books, Washington, D.C., 1964, pp. 443-454.

16. "The Dartmouth Time-Sharing System," Computation Center, Dartmouth College, Oct. 19, 1964.

17. "PDP-6 Time-Sharing Software," Form F-61B, Digital Equipment Corp., Maynard, Mass.

18. W. W. Lichtenberger and M. W. Pirtle, "A Facility for Experimentation in Man-Machine Interaction," this volume.

19. G. J. Culler and R. W. Huff, "Solution of Nonlinear Integral Equations Using On-line Computer Control," *Proceedings of the Spring Joint Computer Conference, 21,* Spartan Books, Baltimore, 1962, pp. 129-138.

20. G. J. Culler and B. D. Fried, "The TRW Two-Station, On-Line Scientific Computer: General Description," *Computer Augmentation of Human Reasoning, Washington, D. C., June 1964,* Spartan Books, Washington, D.C., 1965.

21. "Carnegie Institute of Technology Computation Center User's Manual."

22. J. C. Shaw, "JOSS: A Designer's View of an Experimental On-Line Computing System," *Proceedings of the Fall Joint Computer Conference, 26,* Spartan Books, Washington, D.C., 1964, pp. 455-464.

23. T. M. Dunn and J. H. Morrissey, "Remote Computing—An Experimental System," Part 1; J. M. Keller, E. C. Strum and G. H. Yang, Part 2, *Proceedings of the Spring Joint Computer Conference, 25,* Spartan Books, Washington, D.C., 1964, pp. 413-443.

24. J. I. Schwartz, "Observations on Time-Shared Systems," *ACM Proceedings of the 20th National Conference,* p. 525 (1965).

25. "Time-Sharing System Scorecard, No. 1 (Spring 1965)," Computer Research Corp., 747 Pleasant St., Belmont, Mass.

26. R. M. Fano, "The MAC System: The Computer Utility Approach," *IEEE Spectrum,* vol. 2, pp. 56-64 (Jan. 1965).

27. P. M. Morse, "Computers and Electronic Data Processing," *Industrial Research,* vol. 6, no. 6, p. 62 (June 1964).

28. T. Kilburn, "One-Level Storage System," *IRE Transactions on Electronic Computers,* vol. EC-11, no. 2 (Apr. 1962).

29. A. W. Holt, "Program Organization and Record Keeping for Dynamic Storage Allocation," *Comm. ACM,* vol. 4, pp. 422-431 (Oct. 1961).

30. J. B. Dennis, "Program Structure in a Multi-Access Computer," Tech. Rep. No. MAC-TR-11, Project MAC, M.I.T., Cambridge, Mass. (1964).

31. J. B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems," *IEEE International Convention Record,* Institute of Electrical and Electronic Engineers, New York, 1965, Part 3, pp. 214-225.

32. J. B. Dennis and E. L. Glaser, "The Structure of On-Line Information Processing Systems," *Information Systems Sciences: Proceedings of the Second Congress,* Spartan Books, Washington, D.C., 1965, pp. 1-11.

33. J. W. Forgie, "A Time- and Memory-Sharing Executive Program for Quick-Response On-Line Applications," this volume.

34. M. N. Greenfield, "Fact Segmentation," *Proceedings of the Spring Joint Computer Conference, 21,* Spartan Books, Baltimore, 1962, pp. 307-315.

35. "The Descriptor," Burroughs Corp., 1961.

36. "Univ. of Mich. Orders IBM Sharing System," *EDP Weekly,* vol. 6, no. 5, p. 9 (May 24, 1965).

37. B. W. Arden et al, "Program and Addressing Structure in a Time-Sharing Environment" (submitted for publication).

38. *Computing Report for the Scientist and Engineer,* Data Processing Division, I.B.M. Corp., vol. 1, no. 1, p. 8 (May 1965).

39. W. T. Comfort, "A Computing System Design for User Service," this volume.

40. "Proposed Revised American Standard Code for Information Interchange," *Comm. ACM,* vol. 8, no. 4, pp. 207-214 (Apr. 1965).

41. P. A. Crisman, ed., *The Compatible Time-Sharing System:* A Programmer's Guide, 2nd ed., M.I.T. Press, Cambridge, Mass., 1965.

42. A. L. Samuel, Time-Sharing on a Computer, *New Scientist 26,* 445 (May 27, 1965) 583-587.

SYSTEM DESIGN OF A COMPUTER FOR TIME SHARING APPLICATIONS*

E. L. Glaser
Massachusetts Institute of Technology
Cambridge, Massachusetts
and
J. F. Couleur and G. A. Oliver
General Electric Computer Division
Phoenix, Arizona

INTRODUCTION

In the late spring and early summer of 1964 it became obvious that greater facility in the computing system was required if time-sharing techniques were to move from the state of an interesting pilot experiment into that of a useful prototype for remote access computer systems. Investigation proved computers that were immediately available could not be adapted readily to meet the difficult set of requirements time-sharing places on any machine. However, there was one system that appeared to be extendible into what was desired. This machine was the General Electric 635. The 635 is a single address stored program computer with a word length of 36 bits. It possessed many of the characteristics that were deemed necessary for the application of a computer to time-sharing. The three most important characteristics are:

1. A clean and comprehensive order code,
2. a multiprocessor capability, and
3. nonsynchronous design.

The first of these requirements stems from the quantity of software to be written for the machine. The size of the operating system demands it be written in some higher level language. An orderly instruction set is essential to permit the use of good code selection algorithms in the compiler. The multiprocessor characteristic was desired to permit a feasible fail-soft characteristic in the system and to allow system growth without major increments equivalent to entire system duplication. The third characteristic, non-synchronous communication between major components, was deemed desirable because of the flexibility afforded in the significant modification needed to achieve the time-sharing system that we had in mind. The nonsynchronous characteristic allows a system to become large without suffering measurable degradation. These modifications in a fully synchronous system could result in degradation of performance. (Degradation of

*Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

course will take place in any system if the delay time of a signal through a cable is a significant part of a basic operating time. However the nonsynchronous approach permits this time to be minimized.)

THE SCOPE OF EXTENSION

The design of the extensions to the 635 began in early May 1964 and the end result is what is now known as the GE 645. The changes are in several areas. First, a totally new I/O control unit has been designed to integrate the control of standard peripheral devices and various types of communications lines. The latter are necessary in the time-sharing environment. A large movable head disc, the DS 25, was available as a standard 635 peripheral. This unit appeared suited for the type of use we envisioned. A high-speed drum (DS 300) was also available, but its performance was not sufficient for the purposes of the highest speed secondary store in the projected time-sharing system. Therefore, a new high-speed drum system was designed for this function (MS 32). The introduction of a new form of addressing logic incorporating segments and pages is a significant change to the system. This, with its concomitant changes in interrupt logic and related portions of the machine affected by it, was by far the major change to the system.

THE SEGMENTS AND PAGES

The concept of the paged memory has appeared in the literature for the past several years and has been implemented on at least one machine[1,2]. The purpose of paging is to make the allocation of physical memory easier. One can think of paging as the intermediate ground between a fully associative memory, having each word addressed by means of some part of its contents, and a normal memory, having each memory location addressed by a specific integer forever fixed to that physical location. In paging, blocks of memory are assigned differing base addresses. Addressing within a block is relative to the beginning of the block. Thus if association and relative addressing are handled with a break occuring within a normal break of the word (viz. in a binary machine block size is a power of 2), then a number of noncontiguous blocks of memory can be made to look contiguous through proper association. The association between a block and a specific base address can be dynamically changed by program during the execution of appropriate parts of the executive routine.

Segments on the other hand are used not for the allocation of physical memory, but for the allocation of address space. The concept of segments has received little attention in the literature until recently.[3,4,5,6] A segment defines some object such as a data area, a procedure (program) or the like. In a sense, each segment corresponds to a virtual memory whose size is whatever size, up to a maximum limit, that is required. In theory, as many such segments can be available to a programmer as necessary. In the case of the 645, the practical limit is 2^{18} segments, each one of which can obtain up to 2^{18} words. Observe that although segments and pages are two distinctly different entities, they work together to facilitate the allocation of physical memory and virtual memory. Although a large number of segments may be defined, each one having a large number of words, only the currently referenced pages of pertinent segments need to be in memory at any time. A very limited concept of segments has been used in computers previously.[7] Recently other system designs have employed a similar segment and paging technique.[8]

DESCRIPTORS

A descriptor is a word that is used to define and locate in physical memory either a page or a segment. Hence there are two kinds of descriptors: page descriptors and segment descriptors. The difference between them is the table in which they are found. Each has slightly different functions as will become obvious.

A segment descriptor contains among other things the location of either the segment itself or, if the segment is paged, the location of the table in which its pages are defined. Each page descriptor corresponds to one of the pages of the segment. A page descriptor contains the location of the base of the block of memory in which this page is to be found. All of the page descriptors for a given segment must lie in contiguous locations of the page table for that segment.

Both segment and page descriptors also contain certain access control information known as the descriptor control field. These fields define the nature of access permitted to a particular piece of information. An example of such a control might be the Write Permit Bit. This bit determines whether

this segment or this particular page of a segment can be written into or only read. Alternatively, we may think of the segment descriptor as defining a certain set of restrictions on accessing the entire segment. Specific page descriptors may add additional restrictions; however, they may not take any away. For example, the segment is defined as being a data segment with writing permitted. The result is that control can not be transferred to this segment, but the words of the segment can be used as data with either reading or writing possible. If for some reason there is one specific page of data that we wish to protect, that page can be marked as "Read Only" (the write permit bit is set to zero). As a consequence, this page is defined not only as data but it is now "Read Only" data. The page descriptor has applied an additional constraint above and beyond those constraints contained in the segment descriptor. In addition to the control and address information the segment descriptor contains a bounds field. This bounds field defines the number of pages that the segment contains. In the case of unpaged segments this bounds field defines the total number of words in the segment.

THE DESCRIPTOR SEGMENT AND BASE REGISTERS

The segment descriptors associated with a given process are all contained in a single segment known as the descriptor segment. The descriptor segment has a distinguished role in the operation of the system in that the processor uses this segment as the sole means of relating program references to memory location. The descriptor segment may be paged in the same manner as any other segment. Its location in memory is defined by a special processor register known as the descriptor base register. This register defines either the base of an unpaged descriptor segment or the base of the segment page table of a paged descriptor segment. This register can only be loaded or stored by privileged instructions not available to slave mode programs. Note that all user programs and most of the executive system are written in slave mode.

A segment can now be identified by an ordinal number which locates its descriptor relative to the base of the descriptor segment. This number is known as the segment number. The address of a location in memory is specified in terms of a segment number and a location within that segment. In the 645 both quantities are expressed as 18 bit numbers. During all addressing in the 645 both parts are necessary except under very unusual circumstances. Both parts are usually supplied explicitly. In some cases they are implied by certain conditions of the machine.

There are several provisions for forming these two-part addresses. The first is by means of the instruction word. An address may refer to a location within the current procedure segment or alternatively to some other segment. A control field in the instruction specifies the choice. If the reference is within the current procedure segment the segment number is found in the procedure base register. This is an internal processor register and not directly accessible to the user. If a reference is to some other segment the segment number is located in one of eight address base registers. The three most significant bits of the instruction address field are used to specify this selection.

A set of commands is available to load, store and modify the contents of the address base registers. Additional flexibility is provided by allowing these base registers to operate as index registers (internal bases). When employed in this manner the indexing base register is coupled with a base register that holds a segment number. This base pair in effect defines a base location internal to the segment. The association of bases, together with marking bases as internal, and "locking" certain bases so that they cannot be changed except in a privileged "master" mode, are all contained in a control register for the bases.

For high-speed storing and reestablishing of status, it is possible to store or load the eight base registers with a single command. Any bases which are locked, on a load bases command are passed over and remain unchanged. The use of these various forms of addressing would be difficult to discuss at this time and the reader is referred to one of the subsequent papers in this session describing the software for the new Multics system.[9]

A second form of segment addressing is made available by means of the indirection facilities in the 645. Two variants are provided. The first of these is known as INDIRECT TO SEGMENT (ITS). This form of modifier requires two words, the first of which gives the segment number together with the tag that indicates it is an ITS word. The second word appears as a normal indirect word in either the 635 or 645. It contains an address to-

gether with an address tag indicating whether further indirection is to take place and if so what type, and additionally if indexing is to take place before using this address. The second variant is by means of the indirect word pair which is called ITB, that is, INDIRECT THROUGH BASE. This form of indirection is identical to the ITS form with the exception that the first word of the pair contains the ITB modifier and a number from 0 through 7 which indicates which of the 8 address base registers contains the segment number of this address. When either form is encountered during indirection, the segment name then in effect is canceled and replaced with that given by the pair. The indirection will continue as prescribed by the pair.

THE THREE MODES OF PROCEDURE EXECUTION

In the 635 there were two modes for program execution: namely, master and slave. In slave mode only a restricted set of processor instructions are executable. Certain instructions such as I/O connect, those instructions dealing with the loading of the elapsed-time register and instructions affecting the relocation register were trapped. In master mode these privileged instructions could be executed and the relocation feature disabled.

In the 645 three distinct modes of execution are defined. These are absolute, master and slave. Slave mode is considered to be the normal mode of instruction execution. In this mode no privileged instructions may be executed. Further the relocation logic for segments and pages is fully operative. Master mode uses the segment and paging hardware identically to slave mode with the exception that privileged instructions may be executed and certain constraints on the access to segments are removed. Absolute mode is superior to the other two modes of operation. In the absolute mode, the segmentation and paging hardware is disabled and all instructions in the machine may be executed. Additionally, none of the segment access restrictions apply. Absolute mode is entered only by the occurrence of an interrupt. The machine enters temporarily into absolute mode to record system status but can be caused to remain in this mode if desired. The segmentation hardware can be temporarily enabled on any instruction merely by the use of the instruction word control bit used to indicate base register selection. Further, encountering either an ITS or ITB modifier will cause the segmentation hardware to be turned on for this instruction execution. When in absolute, the mode of the program can be turned back to either master or slave by the execution of an appropriate instruction. This instruction is a branch instruction that defines a segment number which indicates what form of procedure shall be executed, that is, master or slave.

Because it was felt desirable to make it possible to branch easily between various programs including between slave and master programs, a certain degree of insurance has to be built into the hardware to guarantee that spurious branches would not take place into the middle of master mode programs from slave programs. As a consequence, a master mode procedure when viewed from a slave mode procedure appears to be a segment which can neither be written nor read. Further, the only method of addressing this segment that is permitted is a branch to the 0^{th} location. Any attempt to get at other locations by branch, execute, return or any other instructions will result in an improper procedure fault causing an appropriate interrupt. A special form of procedure called EXECUTE ONLY has also been defined which is similar to MASTER PROCEDURE in terms of entry restrictions imposed on slave mode. Once entered, this procedure has all of the execution characteristics of slave mode.

THE ASSOCIATIVE MEMORY

The addressing system as now defined would be very unsatisfactory if employed for each instruction or operand reference. If both the descriptor segment and the data segment are paged and if the procedure being executed is paged, a large number of memory cycles might be required to develop a memory address. To overcome this an associative memory is incorporated in the processor. This memory "captures" a compounded descriptor, derived from the segment and page descriptors. The resultant working descriptor represents a particular page of a particular segment. This can be either a data segment or the procedure segment. As a consequence, if a particular page of a segment is being used quite heavily, its descriptor will always be in this associative memory and no additional references to main memory are required to develop the memory address. The associative memory is "invis-

ible" to the user. Its only effect is to greatly speed up the execution of the programs. Whenever a new working descriptor is created, it is placed in the associative memory. If the associative memory is already full, a wired algorithm selects a memory position to be used for the new descriptor and causes an older descriptor to be discarded. A set of commands permit storing selected words from the associative memory and clearing the associative memory. All of these instructions are privileged.

ADDITIONAL AIDS TO MEMORY ALLOCATION

The environment in which this system is to work places a high premium on efficient management of memory resources. Paging of itself simplifies the allocation process. A further gain is possible if one appreciates the effect of frequency and duration of usage. Two distinct mechanisms are employed to supply this information. The first of these involves the page descriptors. A record is made in the descriptor if the page is accessed for any reason. Once this bit has been set to a one, it is unaffected by subsequent page references. A supervisor program will periodically reset these "use bits" to zero and at the same time determine which pages have been accessed since the last entry into this procedure. A second bit in each of the page descriptors is set to one if the contents of the page is altered in any way.

The second mechanism involves the associative memory. The privileged instructions that store the contents of the entire associative memory or store the contents of the cell whose contents are "the oldest" descriptor provide the supervisory program with a measure of the extent and frequency of page usage.

INTERRUPT CONSIDERATIONS

In the 645 interrupts are generated by external stimuli while faults are generated by processor conditions. The occurence of either an interrupt or fault causes the execution of two commands located at specific "wired" addresses. On most computers, interrupt can only take place between command executions. In the case of the 645, certain aspects of segmentation made desirable the interruption of the computer at many points during execution of a specific command. After the interrupt is serviced, execution is resumed. Resumption at the precise point of interruption, rather than restart, is mandatory because of the nature of the indirection in the 645.

The features of the segmentation system which first made it mandatory to add this more generalized interrupt capability were associated with the various control checks implied by the descriptors. Examples of these control checks are the bounds check, attempting to write in a Read Only segment, etc. It is advantageous in a segmented environment to cause an interrupt as a result of accessing an appropriately coded segment or page descriptor. This type of interrupt or fault is called a Directed Fault. Its name is derived from the fact that this descriptor directs control to a specific function based on which one of 8-bit configurations is found in the segment or page descriptor. The encoding and placement of this type of descriptor is done by the supervisor. The use of these descriptors for marking missing pages or missing segments will be discussed in a subsequent paper.

The 645 interrupt handling mechanism has been called the "snapshot" register. This is a set of flip-flops which, although used by other functions of the machine, are primarily available for storing the machine state. A trap, be it either an interrupt or a fault, causes the state of the machine to be stored in this snapshot register. The contents of essential registers and a history of control states comprise the snapshot. The first instruction in the interrupt handling routine normally will be a store control unit instruction. This privileged instruction stores the contents of the snapshot register into six memory locations. The subsequent execution of a restore control unit instruction takes the contents of the six words and reestablishes the control unit.

CONFIGURATION CONTROLS

Because of the highly on-line nature of this system it is necessary to reconfigure the system relatively easily. As a consequence, those switches required for reconfiguration control are remotely located from the units they control to allow rapid setting from a central point. At first glance it might seem desirable to make program configuration possible; however, if the system is malfunctioning it would be necessary in any event for the program to notify and probably obtain permission from the floor supervisor. As a consequence, it was decided initially to make reconfiguration controllable by manual switches and to

allow the computer to instruct the operator during reconfiguration.

Reconfiguration is used for two prime purposes: to remove a unit from the system for service or because of malfunction, or to reconfigure the system either because of the malfunction of one of the units or to "partition" the system so as to have two or more independent systems. In this last case, partitioning would be used either to debug a new system supervisor or perhaps to aid in the diagnostic analysis of a hardware malfunction where more than a single system component were needed.

The effectiveness of rapid reconfiguration is difficult to determine in a paper simulation and efficiency of the system chosen will only be proved or disproved after a number of months of practical use in the one-line environment.

CONCLUSIONS

We have attempted in this paper to describe some of the considerations that led to the unique design of the GE 645 as a processor for a multi-access, remote user, information processing system. We have already learned much in the process of designing this machine and feel that within the next two to three years much more will be learned. It is difficult at this time to make any statements about what the future of such processors should be beyond a few tentative conclusions.

First, additional speed both in arithmetic capability and in the memory hierarchy is desirable still; but even more, increased channel capacity of the main store of the machine is required. At present it appears that the principal limitation and expansion of the system will be the channel capacity of core memory. Obviously, this will be alleviated to a great extent by the advent of higher speed central memories for computers. However, this is but one answer. We feel that superior facilities can be gained by closer attention to the system functions that we have emphasized with the 645; namely efficient interrupt handling capability, and comprehensive addressing logic to improve the allocation and protection of physical and logical memory.

Finally, it is felt that the designer of future time-sharing systems must remember that a main part of the system is not in the computing center. Rather it is composed of the communications lines, the terminals and the various users, be they human beings, experiments or the like at these terminals. Therefore, the designer must keep in mind that he is engaged in a communications activity as well as an information processing activity and that proper attention must be paid to both aspects. We have done this to the best of our ability in the present system, although we are sure that some several years from now we will be able to return with the description of a machine that will be as great a step over the 645 as the 645 is over previous designs when applied to this new emerging field of time-shared computation.

REFERENCES

1. F. H. Sumner, "The Central Control Unit of the Atlas Computer," *Proc. IFIP Congress*, 1962, pp. 291-296.

2. J. Fotheringham, "Dynamic Storage Allocation in the Atlas Computer," *Comm. ACM*, vol. 4, no. 10, Oct. 1961, pp. 435-436.

3. A. W. Holt, "Program Organization and Record Keeping for Dynamic Storage Allocation," *Comm. ACM*, vol. 4, no. 10, Oct. 1961, pp. 422-431.

4. M. N. Greenfield, "Fact Segmentation," *Proc. SJCC*, vol. 21, May 1962, pp. 307-315.

5. J. B. Dennis and E. B. Van Horn, "Nesting and Recursion of Procedures in a Segmented Memory," Project MAC Memo, M-187, M.I.T., Oct. 1964.

6. J. B. Dennis and E. L. Glaser, "The Structure of On-Line Information Processing System," *Proc. of the Second Congress on Information System Sciences*, Spartan Books, Inc., Washington, D.C., 1965.

7. "The Descriptor," Burroughs Corp., 1961.

8. "IBM 360, Model 67, Computing Report for the Scientist and Engineer," *1*, 1 (May 1965) p. 8, Data Processing Division, I.B.M. Corporation.

9. V. A. Vyssotsky, F. J. Corbato, R. M. Graham, "Structure of the Multics Supervisor," this volume.

STRUCTURE OF THE MULTICS SUPERVISOR*

V. A. Vyssotsky
Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey
and
F. J. Corbato, R. M. Graham
Massachusetts Institute of Technology
Cambridge, Massachusetts

INTRODUCTION

This paper is a preliminary report on a system which has not yet been implemented. Of necessity, it therefore reports on status and objectives rather than on performance. We are impelled to produce such a prospectus by two considerations. First, time-sharing and multiprogramming are currently of great interest to many groups in the computing fraternity; a number of time-sharing systems are now being developed. Discussion of the issues and presentation of goals and techniques is valuable only if it is timely, and the appropriate time is now. Second, every large project undergoes a subtle alteration of goals as it proceeds, extending its aims in some areas, retracting them in others. We believe it will prove valuable to us and others to have on record our intentions of 1965, so that in 1966 and 1967 an unambiguous evaluation of our successes and failures can be made.

The scope of this paper is an operating system in the strict sense. It is only slightly concerned with the hardware of the GE 645, for which the system is now being implemented. It is equally little concerned with the translators and utility programs which make the system useful for computing. Furthermore, this paper pays little attention to the file system, which is the largest single component of the operating system, including well over half of the total code. A separate paper is devoted to the file system.

Much of the content of this paper is statements of mechanisms or techniques for achieving particular goals. In very few cases do we discuss proposed alternative methods, or our reasons for choosing particular methods. Such discussion would require an extended treatise; such a treatise might be useful, but it does not exist, and is not likely to. We hope to produce fragments of it in the future. In every case, our choice of method is based on one or more of four criteria. First, some of the mechanisms were adopted from previous systems because they proved satisfactory there. Second, alternative solutions to some of the problems were tried on previous systems and found unsatisfactory. Third, in some cases the merits and defects of alternative

*Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

methods have been vigorously debated and subjected to gedanken experiments; the chosen method was that which appeared most satisfactory (or least unsatisfactory). Finally, many approaches were chosen because they are evidently workable and are well aligned with the overall approach advocated by our firmly opinionated planning group. The Strongest opinion of our planning group is that consistency is a virtue, and that general solutions are better than particular ones.

VIEWPOINTS AND OBJECTIVES

We view an operating system as an evolving entity. Every operating system with which we have been associated has been greatly modified during its useful life. Therefore, we view the initial version of Multics not as a finished product to be cast in concrete, but as a prototype to be extended in the future. In two ways this is an unhappy conclusion. Users (except those users who benefit substantially from a particular change) tend to resent bitterly any fluidity in the tools with which they must work. System programmers become satiated with reworking programs which they would like to forget. However, the one thing which most users resent more than a fluid system is a frozen system inadequate to the users' expanding needs. So the system must evolve.

Therefore, one of the primary objectives of Multics is that it shall include any features that we can clearly discern to be useful in allowing future changes or extensions to be made with minimum effort and minimum disruption of existing applications. The initial cost of including such features is substantial. We believe from past experience that the initial cost will be more than repaid in reduced future cost of reworking both the operating system and the application programs that use the system.

We view the operating system as having an ill-defined boundary. The software field is replete with examples of user installations or individual application programmers using a cutting torch and jack hammer to break into a neatly defined software package. The effort involved in many such cases is so large as to constitute prima facie evidence that the job was not done for frivolous reasons.

Therefore, Multics is designed to be a single-level system. Most modules of the operating system itself are indistinguishable from user programs, except that they are guarded against unintended or ill-advised changes by protective locks administered by the user installation. Changes to the operating system can therefore be made by the same techniques as are used to change user programs. A programmer who wishes to change a module of the operating system must be authorized to do so. He does not, however, need a large "system edit" program, since the format and conventions of operating system modules are the same as those of user programs.

We view a large open-shop computer facility as a utility like a power company or water company. This view is independent of the existence or non-existence of remote consoles. The implications of such a view are several. A utility must be dependable, more dependable than existing hardware of commercial general-purpose computers. A utility, by its nature, must provide service on demand, without advance notice in most cases. A utility must provide small amounts of service to small users, large amounts to large users, within very wide limits. A utility must not meddle in its customers' business, except by their request. A utility charges for its services on some basis closely related to amount of service rendered. A utility must provide its product to customers more cheaply or more conveniently than they could supply it for themselves. Most important of all, a utility must provide service to customers who neither know nor wish to know the detailed technology employed by the utility in providing the service.

All of these considerations save played a role in the design of Multics. The file system contains elaborate automatic backup and restart facilities to make the dependability of information storage within the system greater than the dependability of the media on which the information is recorded. The operating system is designed to be dynamically adjustable to compensate for temporary loss of one or more hardware modules. Multics is designed to provide service without batching or prescheduling, although prescheduling facilities will be provided for runs whose size and urgency dictates such treatment. Multics employs allocation and scheduling algorithms intended to allow small and large jobs to flow through the machine together, without differentiation, with any special priorities supplied by human beings on the basis of urgency of jobs (or categories of jobs), rather than built-in priorities based on size or type of job. An explicit criterion of Multics is that computation center personnel shall not be required to take cognizance of, or perform any action whatsoever for, a routine job which

does not demand unusual facilities. Multics is intended to accommodate within it standard (but replacable) charging and accounting routines. Multics will accommodate a variety of input-output terminals, ranging from Teletypes to line printers to laboratory measuring equipment for the convenience of its users. The scheduling and allocation algorithms are intended to run the installation with low housekeeping overhead, especially when the load is heavy.

The most important consideration is the one which Multics seems least likely to meet to the satisfaction of its designers. Most of the ultimate users of a large-scale computer have no interest whatsoever in computers or computer programming, let alone the details of particular machines, programming language and operating systems. They have problems to which they wish answers, or data they wish transformed or summarized in some particular way. No computer shop can be considered to function satisfactorily as a utility unless the users can get results without having to formulate the problems in an alien notation. In other words, the system should be sympathetic to its users. Multics provides no direct assistance toward this goal, and little indirect assistance. Neither can any amount of evolution of algebraic languages offer much assistance, since they are still programming languages closely reflecting the structure of a digital computer, and most users are not interested in programming computers in the first place. Progress in this area will require extensive effort in analysis of particular application fields, and development of specialized program packages relevant to the specialized needs of the application fields. The only assistance Multics provides is a framework within which a user can conveniently interact with such a specialized package if it exists, and a measure of isolation from detailed hardware eccentricities which should very substantially ease the life of programmers developing such packages.

We consider privacy of user information to be vitally important. In many applications it is essential that all authorized personnel, and no unauthorized personnel, should have easy access to programs and data. Multics provides, in its hierarchial file structure and its protection mechanisms, very substantial aids to privacy. These aids, when intelligently used, should provide virtual certainty that unintentional privacy violations will not occur, and should provide excellent protection against intentional, ill-advised, but unmalicious attempts to access or modify private information without permission. Multics does not safeguard against sustained and intelligently conceived espionage, and it is not intended to.

ADMISSIBLE HARDWARE CONFIGURATIONS

The minimum hardware configuration with which 645 Multics can run is one 645 CPU, 64K of core memory, one high-speed drum or one disc unit, four tape units, and eight typewriter consoles. However, Multics will not run efficiently on this minimum configuration, and would normally be operated thus only when a substantial part of a larger configuration was unavailable for some reason.

A small but useful hardware complement would be 2 CPU units, 128K of core, 4 million words of high speed drum, 16 million words of disc, 8 tapes, 2 card readers, 2 line printers, 1 card punch and 30 consoles.

The initial implementation of 645 Multics software is designed to support a maximum configuration of up to 8 CPU's, up to 16 million words of core, up to 2 high speed drums, up to 300 million words of disc and disc-like devices, up to 32 tapes, up to 8 card readers, 8 punches, 16 printers, and up to 1000 or more typewriter consoles. It will not, of course operate efficiently (or in some cases at all) with an arbitrary and unbalanced mixture of these. For instance, 645 Multics would not run well with 6 CPU's and 128K words of core.

TECHNICAL POLICY FOR WRITING SOFTWARE

As stated earlier, Multics is intended to be a single level system, and an evolving system. In spite of evolutionary tendencies, 645 Multics must be a useful product and it is to be in operational use in 1966. These factors combine to motivate a small but crucial body of technical policy for system programming. This technical policy differs from standards of good practice in that technical policy is mandatory and enforced upon system programmers working on 645 Multics, and requests for exceptions are skeptically reviewed by project supervision.

Absolute mode (execution without relocation of addresses) is used only

a) for the first two instructions of each trap-answering routine
b) for startup of a cold machine
c) for the initial stages of catastrophe recovery (e.g., recovery from a trouble fault), and
d) for appropriate product service routines (hardware test and diagnostic routines).

Master mode (execution with unrestricted access to privileged hardware features) is used only

a) for absolute mode execution
b) to exercise privileged hardware features
c) where temporary disabling of all interrupts is required, and
d) for appropriate product service routines.

Code which is written in master mode because its purpose is to exercise privileged hardware features will be written as standard subroutines. Each such subroutine may perform only one function (e.g., issue an I/O select). Each such subroutine will check the validity of the call.

All operating system data layouts for the initial implementation of 645 Multics will be compatible with data layouts used by PL/I, except where hardware constraints dictate otherwise. All modules of the initial implementation of the operating system will be written in PL/I, except where hardware constraints make it impossible to express the function of the module in the PL/I language.

All procedures and data will be usable paged to 64 words, paged to 1024 words, or unpaged, except for vectors and data blocks which are inherently unpaged because of direct hardware access to them.

Since the PL/I translator which will be used until mid-1966 generates inefficient object code, it is clear that 645 Multics in its first few months of existence will be inefficient. This penalty is being paid deliberately. After mid-1966, two courses of action will be available: upgrade the compiler to compile more efficient code, or recode selected modules by hand in machine language. We expect that both strategies will be employed, but we expect to place preponderant emphasis on upgrading the PL/I compiler; indeed, one subsequent version of PL/I is already being implemented, and a second is being designed.

PROCESSES

In Multics the activities of the system are divided into *processes*. The notion of process is intuitive, and therefore slightly imprecise. To convey the notion we shall talk around it a bit, and then give a reasonably exact definition.

When a signal from the external world (e.g., a timer runout signal) arrives, and a CPU interrupt occurs, what is being interrupted? Presumably a "run." Observe that if a program is defined in the usual way as a procedure plus data, there is no meaning to the phrase "interrupt a program," if it is taken literally. What is interrupted is the execution of a program. In a time-sharing system this distinction becomes so important, and ignoring the distinction is so pernicious, that we shall use the word "process" to denote the execution of a program, and reserve the word "program" to denote the pattern of bits (or characters) which the hardware decodes.

In most cases a process corresponds to a job, or run; it is a sequence of actions. Consider for example the sequence of actions: build a source program, compile it, execute it and the programs it requires, produce output files including postmortem information and accounting data. This sequence of actions would typically be a single process in Multics.

If the notion of process is to be useful, it must be possible, given some action, to determine to which process it pertains; that is, it must be possible to distinguish unambiguously between processes. In 645 Multics we base our distinction on descriptor segments. At any given moment a 645 CPU is using one and only one segment as the descriptor segment. At different times the CPU may use various different descriptor segments. We define a process to be all those actions performed by a CPU with some given segment as descriptor segment, from the first time that segment becomes the descriptor segment until the last time the segment ceases to be the descriptor segment. Thus a process has a very definite beginning; if it ends, it has an equally definite end.

For each process there is in addition to the descriptor segment a stack segment, for the user's programs and most supervisory routines, and a concealed stack segment, used by some supervisory routines to hold information such as charging data, which must be safeguarded against garden variety user program errors. There are also any other seg-

ments (including supervisory segments) which are required by the process. For each process there will typically be many segments, containing the user and supervisor programs and data, but most of the segments will be attached to the process only as they are dynamically required.

Since we have already observed that almost no procedures will run in absolute mode, and since the operational definition of process places all master mode and slave mode execution firmly in some process, it follows that almost all CPU activity occurs as part of some process. Most processes will be initiated by customers and charged to customers. Some processes will be initiated by the installation and charged to overhead. An example is a process which purges a disc unit.

STATUS OF A PROCESS

Any process that exists in 645 Multics is either running, ready, or blocked. A process is *running* if its descriptor segment is currently being used as the descriptor segment for some CPU. A process is *ready* if it is not running but is not held up awaiting any event in the external world or in another process. A process is *blocked* if it is awaiting an event in the external world or in another process (e.g., arrival of input data, or completion of output, or 3 PM, or retrieval of a page from drum, or release of a data file by another process).

SEGMENTATION, PAGING AND ADDRESSABLE STORAGE

A general principle in Multics is that programs are written to reference locations in addressable storage, rather than locations in core. An address consists of a segment number and word number. The address of an item is clearly important to the program, and possibly to the programmer. Therefore, in Multics the division of programs and data into segments, and the sizes, names and types of the segments, are controlled (explicitly or implicitly) by customers and customer processes.

Paging, on the other hand, is considered in Multics to be the responsibility of the operating system. The view of the designers of Multics is that provided the customer gets his answers when he wants at the price he expects to pay and agrees to pay, it is none of his business where in core his programs and data resided—nor, indeed, whether they were in core at all. The 645 hardware was designed with this philosophy, and the software is built to implement this approach.

However, in some real-time applications it is demonstrable that the application cannot be correctly implemented unless certain programs and data are in core when external signals arrive. In some other applications reasonable efficiency may be attainable only if the user program can specify explicitly what should be in core at which stages of execution. Therefore, calls to the paging routines are provided for specifying:

a) that certain procedures and data must be "bolted to core" in order for the application to run,
b) that certain material is going to be accessed soon, and should be brought into core if possible,
c) that certain material will not be accessed again, and may be removed from core.

It is expected that few application programs will need to make use of such calls.

The paging routines will normally operate with only three sources of input information.

The pager will know when a page must be brought into core by the fact that a page-not-in-core fault occurs. It will know which pages are candidates to be removed from core by a usage measure it derives from the "used" bit of each page table entry, and by a specification in the core map of whether the page is accessed other than through a page table (e.g., a page which is itself a page table, and therefore is referenced directly by CPU hardware). The pager will also know from specifications in the core map which pages may not be removed from core at all (e.g., because they are currently attached to peripheral devices).

A program such as the linker will deal with addressable storage, and will not consider the place of physical residence of any procedure or data block in establishing a linkage. If the linker happens to access information which is not in core, the pager will be invoked by a page-not-in-core fault, the process in which the linker was working will be blocked until the page arrives, and will then be ready to resume.

SEGMENTS AND FILES

In 645 Multics, every segment is a file, and every file is a segment. A reference to one of these ob-

jects, however, may be made in two distinct ways: by segment referencing and by file referencing. Segment referencing is, by definition, referencing by means of a 2-component numerical address, each component consisting of 18 bits, of which the first component specifies a word number in the descriptor segment and the second specifies a word number in the referenced segment. File referencing is anything else. Every file is a segment to some procedure in some process at some time. Any file reference which results in retrieval or modification of any part of the contents of a file (except retrieval, replacement or deletion of the entire file) is a call to a procedure which references the file by segment referencing. Thus, the question of whether a data object is a segment or a file is a question about the viewpoint from which some particular procedure sees the file.

Segments (files) come in two varieties: bounded segments and unbounded segments. A bounded segment is a segment which is guaranteed to consist of 2^{18} words or less. An unbounded segment may have any number of words (e.g., 27), but is not guaranteed to have no more than 2^{18}. You have to look at it to find out. Segment referencing using the appending hardware can only be done directly for bounded segments. To each unbounded segment there may be associated a bounded segment called a "window"; the origin of the window segment may be set, by a supervisor call, to any 1024 word boundary in the unbounded segment. More than one window segment may be attached to a single unbounded segment, if desired, and the windows may be adjusted independently. In principle, the size of an unbounded segment could be arbitrarily large. However, the software of 645 Multics will limit the size of unbounded segments to 2^{28} words, and in some installations storage limitations will hold the maximum segment size even below 2^{28} words.

PERIPHERAL DEVICES AND FILES

In 645 Multics, one of the kinds of file given special recognition will be the serial file. In 645 Multics, unit record equipment and typewriter-like consoles will be treated as serial files of restricted capabilities. User programs will be able to know that such hardware units are not serial files, but it will not normally be advantageous to make use of that fact, and to use such knowledge may severely restrict the applicability of a program. If a program handling a peripheral device as a serial file attempts to perform an illegal primitive (e.g., rewind a card reader), then either

a) the effect on all ensuing processing will be as if the primitive had been performed successfully (e.g., the input file copied from the card reader will be rewound) or
b) a diagnostic will occur (e.g., skip to the end of file on typewriter input).

The effect of treating peripheral devices as serial files is to make it possible for many programs to run either with a typewriter console as a peripheral device or with the console replaced by files on secondary storage.

SCHEDULING

In Multics the system is regarded as having a pool of anonymous CPU's; scheduling and dispatching procedures are executed by each CPU when it must determine what to do next. The only result with any operational meaning that can ensue from scheduling and dispatching in Multics is that CPU number n resumes process p at time t. Furthermore that process must have been in ready status.

We shall state here some fundamental assumptions concerning scheduling which appear evident to us, but some of which are not universally accepted. The goal of scheduling in an open-shop general purpose computer system is to give good service to customers at reasonable cost. When the offered load is greater than system capacity, it is impossible to give good service to all those who desire it. Therefore, on an overloaded system, scheduling should be done so as to minimize overhead and to complete the most urgent work first. Two basic techniques for minimizing overhead are to employ service denial rather than service degradation, and to minimize the number of times control is switched from one process to another. That is, it is more efficient to serve a few users at a time and do it well than it is to serve all users poorly at once. A job is urgent, in the last analysis, because it is costing someone time and/or money not to have the results. The urgency of a job is only slightly correlated, if at all, with the extent of its demands on such system resources as CPU time, core storage, secondary storage, and peripheral facilities. Hence, urgency of work must be determined by human beings, not by the computer.

If offered load is less than system capacity, it is possible in principle to give good service to all who desire it. It may not be possible, however, to achieve satisfactory service for all and still keep the percentage of overhead low. A moderate increase in overhead on a lightly loaded system is acceptable if the increase permits improved service.

Switching between processes is mandatory when a given process becomes blocked. Switching is done at other times to meet explicit or implied service guarantees. For example, placing a typewriter in a customer's office implies a guarantee that response times to simple requests will usually be short. Therefore, frequent switching between processes makes excellent sense when offered load is light, although not when offered load is heavy.

Offered load will rarely be well-matched to system capacity. Any general-purpose open shop computing installation where offered load is at the same approximate level at 3 a.m. Sunday and 3 p.m. Wednesday is either employing load flattening measures outside the computing shop (e.g., by human prescheduling) or is so heavily overloaded that offered load is almost always above system capacity, and service denial is the rule of the shop.

We believe that a general-purpose open-shop computing facility which is never (or almost never) overloaded is spending too much money for computing hardware. It is cheaper to accept occasional overloads. Further, we believe that any scheduling technique for a time-shared multiprogrammed computer system which behaves satisfactorily during overload will require at most a very slight modification to behave well under light load.

Hence, we contemplate an environment in which offered load is almost always either substantially above or substantially below system capacity. We believe that scheduling algorithms should be designed with good performance during overload as the primary objective, and good performance when load is light as a criterion to be met within the framework imposed by the overload design. The scheduler should get information concerning urgency of jobs from human beings, and should not have any built-in assumptions that console jobs are either more or less urgent than absentee jobs, or that short runs are either more or less urgent than long runs.

Unfortunately, in a multiprogrammed time-sharing system with dynamic storage allocation neither the machine nor human beings can determine directly how large the offered load is. How, for example, could one tell how many people at typewriter consoles would type messages if you unlocked their keyboards? Similarly, it is not possible in most cases to predict with any accuracy what demands a given process will make upon system resources during its next few seconds of running. Therefore, the scheduling algorithm must base its action on measurable quantities related to the unmeasurable offered load.

Several such measurable quantities are conveniently available. The most important of these appears to be a running measure of the rate of progress toward completion of processes, compared with a "satisfactory" rate of progress determined by information supplied by human beings about types of processes or individual processes. For example, if there are exactly six processes to be considered each requiring 20 seconds of CPU time and no I/O, all with desired completion time 3 minutes away, and if in one second each process has received 100 milliseconds of CPU time, then each process at its current rate will require 3 minutes 20 seconds to complete. Presumably the system is overloaded, and one or more of the processes should be postponed. This is a fairly typical example; overloads in a system with dynamic storage allocation tend to become manifest by excessive overhead rather than by excessive visible demand. The scheduling algorithms for Multics will rely heavily on this fact.

The choice of which processes to postpone depends on several factors. If some processes have higher priority than others, the lower priority processes will be postponed. If, in the lowest priority class which will continue to run, some processes have been prescheduled for given completion times or computing rates, the prescheduled processes will be given preference. Finally, to make a choice among processes otherwise equal, the scheduler will prefer a process currently using expensive facilities (e.g., core) over one occupying inexpensive facilities (e.g., drum); the former is in some sense using more system resources than the latter, so it is desirable to move it toward completion.

DYNAMIC LINKING

In Multics linking of one procedure segment to another, or of a data segment to procedures, is by and large done dynamically. That is, if a translator compiles symbolic intersegment references, these will not normally be replaced by numerical interseg-

ment references until the first time the reference actually occurs during execution of the compiled program.

The standard form of programs in Multics will be common shared procedure. Code run as common shared procedure may not be modified for execution of any one process. Hence, for each compiled segment of code there will be an accompanying linkage section, which will be maintained on a per-process basis, and all modifications required to link two segments together will be made in the linkage sections rather than in procedure segments. A linkage section contains, among other things:

(a) the symbolic (character string) name of each externally known symbol within the segment to which the linkage section belongs.
(b) for each symbolic reference from this segment to some other segment, the symbolic name of the foreign segment and the symbolic name of the referent within the foreign segment, plus an indirect word which is compiled with a tag that will cause a trap to occur when an indirection through it is attempted.

When a procedure is attached to a process, the linkage segment of the procedure is copied into a data segment of the process. If the procedure during execution attempts to access a foreign segment by indirection through the linkage section, a trap ("linkage fault") will occur. At this time the linker will substitute the correct numerical value into the indirect word. The reference will then be completed; subsequent references, of course, will be completed without occurrence of a trap.

The original symbolic information is retained in the linkage section even after linking. Hence, it is possible to break such a link after it has been established, and detach a segment from a process. This will be done only upon explicit call to the unlinker, and is expected to be infrequent.

TRAP HANDLING

The hardware traps on the 645 can be divided into two categories. In one category are process traps (e.g., overflow) which normally occur as a consequence of action in the running process. Handling of these traps will be done as part of the running process, by supervisory routines attached to the process. In the other category are system traps, some of which are relevant to some process but probably not one which is running (e.g., I/O termination), and others of which indicate hardware or software error (e.g., parity error in core).

Some of the process traps, such as the illegal procedure fault, will cause the process to be removed from running state after a bit of initial flailing around. The division between process traps and system traps is not based on whether the running process will continue to run, but on whether the running process is known to be responsible for the trap.

What happens when a trap occurs? It varies somewhat, but generally speaking the status of the running process is stored in its concealed stack segment. Then, for system traps only, control switches to a special trap process. Then the concealed stack of the process (trap process for system traps, process which is still running for process traps) is pushed one level, and the appropriate trap-handling procedure is called. The supervisory routines have a standard trap-handling procedure for each trap, which discovers what caused the trap and takes appropriate action. However, for every trap there is at least one point in the trap-handling procedure where control will pass to some other routine in the process if the process is administratively entitled to provide alternative treatment for the trap. The extent to which customer processes can provide nonstandard trap handling is, of course, controlled by the installation, but it will by and large vary from complete freedom (for handling overflow) to very strict control (for handling page-not-in-core faults from the appending hardware).

Many traps will have several intercept points, corresponding to different causes of the trap. It should thus be possible for authorized processes to selectively modify the handling of every process trap. Only a restricted group of people will normally be able to modify handling of system traps, since these affect operation of the entire system. The technique for making the modifications, however, is the same as for process traps.

The work of the system trap process is to discover which processes are responsible for system traps. It must, for example, decode words in the status storage channels of the general I/O controller to find out what device caused an I/O interrupt, and then check status tables to discover which process issued the select that resulted in the interrupt. The

trap process can then bring the process responsible for the trap into ready status for further treatment of the particular interrupt; the trap process is then finished with that particular interrupt.

The process responsible for the interrupt may be a customer process; if not, it is a housekeeping process that behaves like a customer process. This process, when it enters running state, will resume in an interrupt routine exactly analogous to a process trap routine, complete with intercept points.

To a very high degree of approximation, all I/O for a process is handled within the process. This does not imply that I/O for each process is handled independently of I/O for other processes. The programs and tables involved in input and output are for the most part common to all processes requiring a given type of I/O activity, such as input from magnetic tape. These programs and tables, however, are attached to each of the processes which requires them, so that they can be called by normal subroutine calls.

This makes it possible to insert special I/O routines (e.g., for controlling a data line to a special-purpose device) in a particular customer process by taking only two actions: get administrative authorization to call relevant master mode routines and to intercept interrupts in the process, and then link to the I/O routines by calling with a standard call. However, this technique places stringent restrictions on timing-dependent I/O, and virtually eliminates the possibility of certain data-dependent I/O techniques. These restrictions appear to be reasonable in a system like Multics; we see no way to permit complete control of I/O by one user program without danger to other user programs.

CREATION, BLOCKING AND TERMINATION OF PROCESSES

Every process begins by being spawned from some other process. In particular, certain system processes exist for no end except to recognize customers' identification and spawn new processes for the customers. However, any process may spawn others by an appropriate call to the operating system. The call specifies what segments the new process is to share with its parent, what segments it should receive copies of, what segments the new process should not know, and at what point the new process should resume.

A process may go into blocked state for many reasons, such as waiting for 3 p.m., or waiting for a page to arrive in core, or waiting for another process to release a file. In all of these cases, the process will indicate a particular flag which must be reset before the process can resume, and the presumption is that some other process (alarm clock routine in the scheduler, or system trap process, or process holding the file) will be cooperative enough to reset the flag. There is, however, no guarantee whatsoever that the flag will ever be reset.

It would be poor strategy to allow the blocked process to remain in limbo forever. Therefore, each process will have attached to it a maximum time for which it may remain continuously blocked. Multics will provide a default value of this time, but a customer may specify a value other than the default value for any particular process. A procedure in the scheduling process will occasionally scan the task list for processes which have been blocked for more than the allowable time. If one is found, a diagnostic message will be generated and shipped off to the error message file for the blocked process, if that can be found, and also to a standard system file. The blocked process will then be completely removed from the task list and, although its procedures and data are still intact, it will not resume if the condition on which it was waiting becomes satisfied. Human intervention is now required to retrieve it, either to attempt to resume it or to obtain diagnostic information. If such human intervention does not occur, the data segments of the process will eventually be purged from the system.

This is also the chain of events which occurs when a process violates some restriction. If, for example, a process attempts to execute a privileged instruction in slave mode, the standard trap procedure will generate a diagnostic message and then call a standard program to force out any relevant output. The process will then go into blocked state to allow a human attempt for further diagnostics or a fixup. If the attempt is not made, the process will then be removed from the task list, and eventually purged.

Termination of a process may occur in two ways. It may call a procedure in the operating system and say "I am through," or some other process may point at it and say "Get rid of him." The second method is used by the scheduler in disposing of processes which have been blocked for too long a

time. This second method may also be used by customer processes, subject to some restrictions.

Both methods may be employed with two degrees of severity. The process may merely be removed from the task list, or it may be marked as completely dead and subject to immediate purging from the system. In general, modules of the operating system will only remove a process from the task list if troubles occur, so that the customer may have a reasonable chance to come and rummage around in the procedures and data of the process to find out what happened.

PROTECTION AGAINST MACHINE ERRORS

Like all other systems, 645 Multics will suffer from hardware and software failures. The goal of dependable operation can be achieved only if the effect of these failures can be limited. A companion paper discusses methods for safeguarding of data in the file system. Equally important and equally difficult is the problem of keeping the system on the air, or getting it back in a hurry, when a hardware failure occurs. This breaks down into two parts: how to run the system on a crippled machine, and how to share the machine with product service routines. We have no solutions to either problem, but some fragments of solutions are developing.

First, the policy of running the CPU's symmetrically is expressly intended to allow any CPU to be pulled at any time without stopping the system (although pulling a CPU at an arbitrary moment will undoubtedly wreck a particular process and some data files).

Second, the policy of minimizing absolute mode operation is designed to allow the system to resume execution with core banks missing with somewhat less agony than would otherwise be the case, and to allow the system to abandon a core bank with very little effort. I/O calls and fabrication of I/O data control words will be concentrated in a few procedures, with the explicit intent of allowing easy abandonment of a general I/O controller. For installations which can afford the luxury of using less than full core interlace, 645 Multics will provide the ability to pick up the pieces more or less automatically after loss of any one core bank, but this feature will probably not be included in the first version of 645 Multics.

We do not know in general how to make the software cope with a berserk CPU, drum controller or general I/O controller. In 645 Multics such a trouble will undoubtedly require a restart, the magnitude of which will vary greatly depending on exactly what the sick hardware unit did before it was caught.

The problem of coexisting with product service routines will be partly solved by subordinating some product service routines to Multics, and partly by the fact that Multics can easily abandon half the hardware of a large enough system on request, so that product service routines can test the other half. It appears likely, however, that integration of product service routines into Multics will be the most difficult aspect of the project, and the last to be satisfactorily completed.

We have no very useful techniques for protecting the system from software bugs. We are reduced to the old-fashioned method of trying to keep the bugs from getting into the software in the first place. This is a primary reason for programming the system in PL/I, and for insisting that modules of the operating system should conform to conventions for user programs. The 645 lends itself exceptionally well to being driven with repeatable sequences of events, and this will help to find timing-dependent software bugs. But some software bugs will survive; they always do.

ACKNOWLEDGMENTS

It would be nearly impossible to name all those who have participated in formulating the material presented in this paper. All of the authors of the other papers in this group have contributed substantially, as have many others of our colleagues and friends. We are particularly grateful to Dr. E. Woiman of Bell Laboratories for allowing us to paraphrase some of his concise observations about the problem of scheduling.

A GENERAL-PURPOSE FILE SYSTEM FOR SECONDARY STORAGE*

R. C. Daley
Massachusetts Institute of Technology
Cambridge, Massachusetts
and
P. G. Neumann
Bell Telephone Laboratories, Inc.
Murray Hill, New Jersey

1. INTRODUCTION

The need for a versatile on-line secondary storage complex in a multiprogramming environment is immense. During on-line interaction, user-owned off-line detachable storage media such as cards and tape become highly undesirable. On the other hand, if all users are to be able to retain as much information as they wish in machine-accessible secondary storage, various needs become crucial: Little-used information must percolate to devices with longer access times, to allow ample space on faster devices for more frequently used files. Furthermore, information must be easy to access when required, it must be safe from accidents and maliciousness, and it should be accessible to other users on an easily controllable basis when desired. Finally, any consideration which is not basic to a user's ability to manipulate this information should be invisible to him unless he specifies otherwise.

The basic formulation of a file system designed to meet these needs is presented here. This formulation provides the user with a simple means of addressing an essentially infinite amount of secondary storage in a machine-independent and device-independent fashion. The basic structure of the file system is independent of machine considerations. Within a hierarchy of files, the user is aware only of symbolic addresses. All physical addressing of a multilevel complex of secondary storage devices is done by the file system, and is not seen by the user.

Section 2 of the paper presents the hierarchical structure of files, which permits flexible use of the system. This structure contains sufficient capabilities to assure versatility. A set of representative control features is presented. Typical commands to the file system are also indicated, but are not elaborated upon; although the existence of these commands is crucial, the actual details of their specific implementations may vary without affecting the design of the basic file structure and of the access control.

*Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

Section 3 discusses the file backup system, which makes secondary storage appear to the user as a single essentially infinite storage medium. The backup system also provides for salvage and catastrophe reload procedures in the event of machine or system failure. Finally, Section 4 presents a summary of the file system program modules and their interrelationship with one another. The modularity of design enables modules affecting secondary storage device usage to be altered without changing other modules. Similarly, the files are formatless at the level of the file system, so that any changes in format do not affect the basic structure of the file system. Machine independence is attempted wherever it is meaningful.

Sections 2 and 3 are essentially self-contained, and may be read independently of the companion papers (see references 1-5). Section 4 requires a knowledge of the first three papers.

2. THE FILE STRUCTURE AND ACCESS CONTROL

In this section of the paper, the logical organization of the file structure is presented. The file structure consists of a basic tree hierarchy of files, across which links may be added to facilitate simple access to files elsewhere in the hierarchy. Each file has an independent means for controlling the way in which it may be used.

If files are to be shared among various users in a way which can be flexibly controlled, various forms of safeguards are desirable. These include:

S1. Safety from someone masquerading as someone else;

S2. Safety from accidents or maliciousness by someone specifically permitted controlled access;

S3. Safety from accidents or maliciousness by someone specifically denied access;

S4. Safety from accidents self-inflicted;

S5. Total privacy, if needed, with access only by one user or a set of users;

S6. Safety from hardware or system software failures;

S7. Security of system safeguards themselves from tampering by nonauthorized users;

S8. Safeguard against overzealous application of other safeguards.

These safeguards recur in the subsequent discussion. The various features of the file system presented below are summarized in Section 2.4, along with the way in which these features help to provide the above safeguards.

2.1 Basic Concepts

In the context of this paper, the word "user" refers to a person, or to a process, or possibly to a class of persons and/or processes. The concept of the user is rigorously defined in terms of a fixed number of *components*, such as an accounting number, a project number, and a name. (Classes of users may be defined by leaving certain components unspecified.) For present purposes, the only users considered are those who employ the file system by means of its normal calls.

A *file* is simply an ordered sequence of *elements*, where an element could be a machine word, a character, or a bit, depending upon the implementation. A user may create, modify or delete files only through the use of the file system. At the level of the file system, a file is formatless. All formatting is done by higher-level modules or by user-supplied programs, if desired. As far as a particular user is concerned, a file has one name, and that name is symbolic. (Symbolic names may be arbitrarily long, and may have syntax of their own. For example, they may consist of several parts, some of which are relevant to the nature of the file, e.g., ALPHA FAP DEBUG.) The user may reference an element in the file by specifying the symbolic file name and the linear index of the element within the file. By using higher-level modules, a user may also be able to reference suitably defined sequences of elements directly by context.

A *directory* is a special file which is maintained by the file system, and which contains a list of *entries*. To a user, an entry appears to be a file and is accessed in terms of its symbolic entry name, which is the user's file name. An *entry name* need be unique only within the directory in which it occurs. In reality, each entry is a pointer of one of two kinds. The entry may point directly to a file (which may itself be a directory) which is stored in secondary storage, or else it may point to another entry in the same or another directory. An entry

which points directly to a file is called a *branch*, while an entry which points to another directory entry is called a *link*. Except for a pathological case mentioned below, a link always eventually points to a branch (although possibly via a chain of links to the branch), and thence to a file. Thus the link and the branch both *effectively point to* the file. (In general, a user will usually not need to know whether a given entry is a branch or a link, but he easily may find out.)

Each branch contains a description of the way in which it may be used and of the way in which it is being used. This description includes information such as the actual physical address of the file, the time this file was created or last modified, the time the file was last referred to, and access control information for the branch (see below). The description also includes the current state of the file (open for reading by N users, open for reading and writing by one user, open for data sharing by N users, or inactive), discussed in Section 4. Some of this information is unavailable to the user.

The only information associated with a link is the pointer to the entry to which it links. This pointer is specified in terms of a symbolic name which uniquely identifies the linked entry within the hierarchy. A link derives its access control information from the branch to which it effectively points.

2.2 The Hierarchy of the File Structure

The hierarchical file structure is discussed here. The discussion of access control features for selected privacy and controlled sharing are deferred until Section 2.3. For ease of understanding, the file structure may be thought of as a tree of files, some of which are directories. That is, with one exception, each file (e.g., each directory) finds itself directly pointed to by exactly one branch in exactly one directory. The exception is the root directory, or *root*, at the root of the tree. Although it is not explicitly pointed to from any directory, the root is implicitly pointed to by a fictitious branch which is known to the file system.

A file directly pointed to in some directory is *immediately inferior* to that directory (and the directory is *immediately superior* to the file). A file which is immediately inferior to a directory which is itself immediately inferior to a second directory is *inferior* to the second directory (and similarly the second directory is *superior* to the file). The root has level zero, and files immediately inferior to it have level one. By extension, inferiority (or superiority) is defined for any number of levels of separation via a chain of immediately inferior (superior) files. (The reader who is disturbed by the level numbers increasing with inferiority may pretend that level numbers have negative signs.) Links are then considered to be superimposed upon, but independent of, the tree structure. Note that the notions of inferiority and superiority are not concerned with links, but only with branches.

In a tree hierarchy of this kind, it seems desirable that a user be able to work in one or a few directories, rather than having to move about continually. It is thus natural for the hierarchy to be so arranged that users with similar interests can share common files and yet have private files when desired. At any one time, a user is considered to be operating in some one directory, called his *working directory*. He may access a file effectively pointed to by an entry in his working directory simply by specifying the entry name. More than one user may have the same working directory at one time.

An example of a simple tree hierarchy without links is shown in Fig. 1. Nonterminal nodes, which are shown as circles, indicate files which are directories, while the lines downward from each such node indicate the entries (i.e., branches) in the directory corresponding to that node. The terminal nodes, which are shown as squares, indicate files other than directories. Letters indicate entry names, while numbers are used for descriptive purposes only, to identify directories in the figure. For example, the letter "J" is the entry name of various entries in different directories in the figure, while the number "0" refers to the root.

An entry name is meaningful only with respect to the directory in which it occurs, and may or may not be unique outside of that directory. For various reasons, it is desirable to have a symbolic name which does uniquely define an entry in the hierarchy as a whole. Such a name is obtained relative to the root, and is called the *tree name*. It consists of the chain of entry names required to reach the entry via a chain of branches from the root. For example, the tree name of the directory corresponding to the node marked 1 in Fig. 1 is A:B:C, where a colon is used to separate entry names. (The two files with entry names D and E shown in this directory have tree names A:B:C:D and A:B:C:E, respectively.)

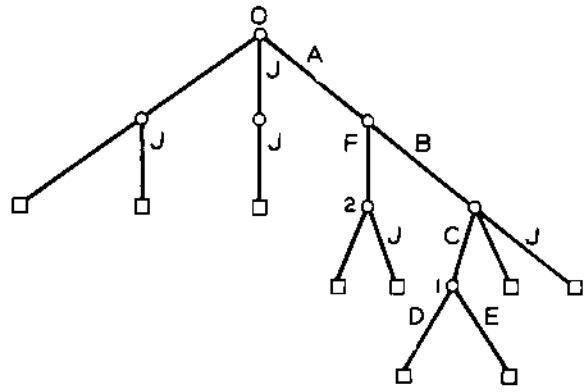

Figure 1. An example of a hierarchy without links.

In most cases, the user will not need to know the tree name of an entry.

Unless specifically stated otherwise, the tree name of a file is defined relative to the root. However, a file may also be named uniquely relative to an arbitrary directory, as follows. If a file X is inferior to a directory Y, the tree name of X relative to Y is the chain of entry names required to reach X from Y. If X is superior to Y, the tree name of X relative to Y consists of a chain of asterisks, one for each level of immediate superiority. (Note that since only the tree structure is being considered, each file other than the root has exactly one immediately superior file.) If the file is neither inferior nor superior to the directory, first find the directory Z with the maximum level which is superior to both X and Y. Then the tree name of X relative to Y consists of the tree name of Z relative to X (a chain of asterisks) followed by the tree name of Y relative to Z (a chain of entry names). For the example of Fig. 1, consider the two directories marked 1 and 2. The tree name of 1 relative to 2 is :*:B:C, while the tree name of 2 relative to 1 is :*:*:F. An initial colon is used to indicate a name which is relative to the working directory.

A link with an arbitrary name (LINKNAME) may be established to an entry in another directory by means of a command

LINK LINKNAME, PATHNAME.

(A command is merely a subroutine call.) The name of the entry to be linked to (PATHNAME) may be specified as a tree name relative to the working directory or to the root, or more generally as a path name (defined below). Note that a file may thus have different names to different users, depending on how it is accessed. A link serves as a shortcut to a branch somewhere else in the hierarchy, and gives the user the illusion that the link is actually a branch pointing directly to the desired file. Although the links add no basic capabilities to those already present within the tree structure of branches, they greatly facilitate the ease with which the file system may be used. Links also help to eliminate the need for duplicate copies of sharable files. The superimposing of links upon the tree structure of Fig. 1 is illustrated in Fig. 2. The dashed lines downward from a node show entries which are links to other entries. When the links are added to the tree structure, the result is a directed graph. (The direction is of course downward from each node.)

In the example of Fig. 2, the entry named G in directory 2 is a link to the branch named C in directory 3. The entry named C in directory 4 (recall that entry names need not be unique except within a directory) is a link to the entry G in directory 2, and thus acts as a link to C in directory 3. Both of these links effectively point to the directory 1.

It is desirable to have a name analogous to the tree name which includes links. Such a name is the *path name*, and is assumed to be relative to the root unless specifically stated otherwise. The path name of a file (relative to the root) is the chain of entry names used to name the file, relative to the root. (For example, the directory 1 in Fig. 2 may have path name A:B:C, A:F:G or H:C, depending on its usage.) The working directory is always established in terms of a path name. A user may change his working directory by means of a command such as

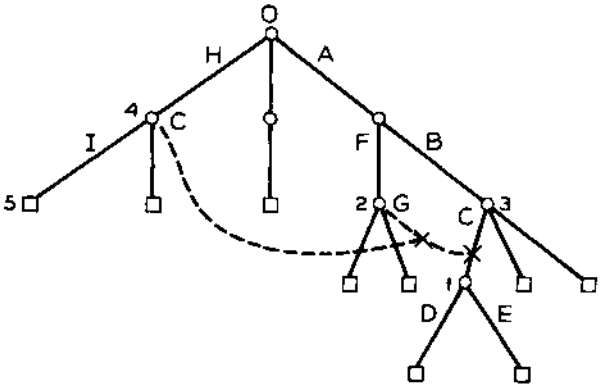

Figure 2. The example of Fig. 1 with links added.

CHANGEDIRECTORY PATHNAME,

where the path name may be relative to the (old) working directory or to the root. The definition of a path name relative to a directory other than the root is similar to the definition of a tree name, with the following exceptions: the concept of a file immediately inferior to a directory is replaced by the concept of a file effectively pointed to by the entry. The concept of a directory immediately superior to a file is replaced by a concept which is well defined only as the inverse of the above effective pointer, that is, dependent on what entry in which directory was previously used to reach the file.

In general, any file may be specified by a path name (which may in fact be a tree name, or an entry name) relative to the current working directory. A file may also be specified by a path name relative to the root. In the former case, the path name begins with a colon, in the latter case it does not.

To illustrate these somewhat elusive concepts, consider the example of Fig. 2. Suppose that the working directory has the path name H (i.e., directory 4). The command

CHANGEDIRECTORY :C

results in the working directory with path name H:C (i.e., directory 1). Subsequent reference to a file with path name :*:I (relative to the working directory with path name H:C) refers to the file 5 in the figure. The command

CHANGEDIRECTORY:*

results in restoring the original working directory with path name H. (With this interpretation of *, the user believes he is working in a tree. Note that the design could be modified so that a path other than the one used on the way down could be used on the way back up toward the root, but not without adding considerable complexity to the design.)

The pathological case referred to above with respect to a link effectively pointing to a file arises as follows. Consider again Fig. 2. Suppose that the branch C in directory 3 is deleted from this directory; suppose also that in the same directory a link with name C is then established to the entry C in directory 4, e.g., by means of the command

LINK C,H:C.

Access to entry C in directory 3 (or to entry G in 2, or C in 4, for that matter) then results in a loop in which no branch is ever found. This and similar loops in which no branch is found may be broken in various ways, for example, by observing whether an entry is used twice on the same access. Note that much more devious loops may arise, as for example that resulting from the establishment of a link (named K) from directory 1 to entry H in the root. Then the path name :C:K relative to directory 4 refers to directory 4 itself. This and similar loops which involve chains of directories are inherent in the use of links, and may in fact be used constructively.

2.3 Access Control

An initial sign-on procedure is normally desirable in order to establish the identity of the user for accounting purposes. It may also be necessary to control the way in which the user may use the sys-

tem. There are two basic approaches to using the hierarchy of files described here. First, the file structure may be *essentially open,* with initial access unrestricted and with subsequent access permitted to all other directories unless specifically denied. On the other hand, the file structure may be *essentially closed,* with initial access restricted for any user to a particular initial directory (assuming his ability to give a password, for example) and with subsequent access to other directories denied unless specifically permitted. There are in fact arguments for each extreme. The essentially open scheme implies that locks need be placed only where they are essential (and most effective). The essentially closed scheme provides well-defined working areas, frees the user from worrying about other users, and helps prevent the user's files from being accidentally altered. It may be observed that the scope of capabilities of the file structure described here does not depend on whether the structure is essentially open or closed. In practice, a position somewhere in between the two extremes is likely to result.

In attempting to access a file, a user may or may not be successful, depending upon what he is trying to do. The basic framework within which permissions are granted is now considered. This framework is independent of the file structure described above. Although the exact set of permissions may therefore vary from system to system, a flexible set adequate for normal usage is given here as an illustration. All permissions are logically on the branches which point to files. (In actual implementation, however, there may in some cases be permissions associated with a directory rather than repeated for each entry in that directory.)

The set of permissions with which a given user may access a particular branch is called the *mode* of the branch for that user. Associated with each branch is an *access control list,* which contains the list of users (or sets of users) along with the corresponding mode associated with each user. The permissions for any users on the list may be overridden (assuming permission to do so—see below) by adding subsequent users and modes to the list. The list is scanned in order of recency, and thus the addition acts as an override. (Each time the access control list is changed, a garbage collection is performed in order to keep the list nonredundant.) All access control information required for the use of a given file is contained in the list on the branch pointing to that file, and is thus independent of the way in which the file was accessed.

The mode consists of five *attributes,* named TRAP, READ, EXECUTE, WRITE and APPEND, each of which is either ON or OFF. In performing access control, the TRAP attribute is examined first. It is by itself powerful enough to accomplish the roles of the other four attributes, which are called *usage attributes.* However, the four usage attributes are included here for ease of description, as well as for ease of use of the system. The four usage attributes indicate permission to perform the given activity on the branch by the particular user only if the corresponding attribute is ON. The function of each attribute is now defined.

TRAP. When a branch has the TRAP attribute ON for a given user, a trap occurs on any reference by that user which affects the contents of the file to which that branch points. In this case, the access control module calls a procedure whose name is given as the first entry of a *trap list.* A trap list may be associated with each user in the access control list. Additional parameters may be defined in the trap list, and are passed as constants to the called procedure. Furthermore, all pertinent information regarding the branch as well as the calling sequence which caused the trap are passed to the called procedure. The traps are processed in the order specified by the trap list. The return to the access control module specifies the effective values of the four usage attributes which are to govern the access. The returned value may override the initial values of these attributes.

The user of a branch may inhibit the trap process. In this case, all references to an entry with the TRAP attribute ON cause an error return to the calling procedure. The TRAP attribute is extremely useful for monitoring of file usage, for placing additional restrictions on access (e.g., user-applied locks), for obtaining subroutines only if and when they are actually referenced, etc. A pair of commands such as LOCK and UNLOCK provide the user with a standard way of applying locks on an entry.

```
LOCK   FILENAME,KEY
UNLOCK FILENAME
```

(FILENAME is the name of a branch given as a

path name.) The command LOCK inserts a trap which on each attempted access may request the user to supply the designated key, and permit access only if the key is correctly supplied. UNLOCK removes the lock. (A timelock command might also be desirable, for example, to make a given branch available to a particular user only between certain times on certain days.) These commands are available to a user only if the branch pointing to the directory which contains the entry FILENAME has the WRITE attribute ON for that user (see below).

The Usage Attributes. The READ, EXECUTE, WRITE and APPEND attributes govern permission to perform operations upon files with certain intents, with an intent corresponding to each attribute. Every operation on a given branch implies one of the four intents, namely *read, execute, write* or *append*. The interpretation of the intent depends upon whether the accessed branch points to a directory (a *directory branch*) or to a nondirectory (a *nondirectory branch*), as seen below.

If a branch is a nondirectory branch, the meaning of each intent is quite simple. The read intent is the desire to read the contents of the file. The execute intent is the desire to execute the contents of the file as a procedure. The write intent is the desire to alter the contents of the file without adding to the end of it. The append intent is the desire to add to the end of the file without altering its original contents. The attribute on a nondirectory branch which corresponds to the particular intent of an operation on that branch indicates permission to carry out that operation only if that attribute is ON.

If a branch is a directory branch, the meaning of each intent is different. The read intent is the desire to read those contents of the directory which may be available to the user, i.e., to obtain an itemization of the directory entries. The execute intent is the desire to search the directory. The write intent is the desire to alter existing entries in the directory without adding new ones. This includes renaming entries, deleting entries, and changing the access control list for branches in that directory. The last of these includes adding traps to the trap list and changing the usage attributes. The append intent is the desire to add new entries without altering the original entries. The attribute on a directory branch which corresponds to the particular intent of an operation on that branch indicates permission to carry out that operation only if that attribute is ON.

Several additional examples of system commands are now given. Assuming the necessary WRITE attributes are ON for the appropriate directory branches, a user may by use of suitable commands change the access control list of entries or delete entries in various ways. For example, he may change (wherever permitted by the WRITE attribute of an inferior branch) the list for all inferior directory branches, or for all inferior nondirectory branches, or for all inferior nondirectory branches whose names include the parts FAP DEBUG, or for all directory branches not more than some number of levels inferior. Similarly, an elaborate delete command may be constructed. (The possibility of no one having the WRITE attribute ON for a given directory branch can be combatted in various ways. One way is not to permit a change in the list to occur which brings about this circumstance, another way is to make this condition imply no restriction.)

Assuming that the necessary READ attributes are ON for the appropriate directory branches, a user may obtain an itemization of desired portions of desired inferior directories, possibly obtaining a graphical picture of the hierarchy.

2.4 Summary of File System Features

At this point, it is desirable to summarize the various features of the file system, and to state which of these contribute to which of the safeguards S1 through S8 mentioned above. The basic features of the file system may be stated as follows:

F1. The inherent hierarchical structure of the file system itself;

F2. The access control which may be associated with a directory branch;

F3. The backup procedures (discussed in the next section).

In addition, certain aspects of the hardware and of the central software also contribute to providing these safeguards:

F4. The hardware, and central software system.[2,3]

The ways in which the safeguards S1-S8 interact with the features F1-F4 are summarized in Table 1.

Table 1. The Features of the File System and Which Safeguards They Assist in Providing.

Safeguards / Features	MASQ S1	PERM S2	DENY S3	SELF S4	PRIV S5	BUGS S6	TAMP S7	ZEAL S8
F1. Hierarchy	Y		Y	Y	Y	Y		
F2. Attributes	Y	Y	Y	Y	Y	Y	Y	Y
F3. Back-up	Y	Y		Y		Y		
F4. System			Y		Y	Y	Y	

Note: Y = YES, the feature does assist.
Blank = NO, it does not.

3. SECONDARY STORAGE BACKUP AND RETRIEVAL

One important aspect of the file system is that the user is given the illusion that the capacity of file storage is infinite. This concept is felt to be extremely important, as it gives all responsibility for remembering files to the system rather than to the individual user. Many computer installations already find themselves in the business of providing tape and card-file storage for their users. It is intended that most of this need will be replaced by the file system in a more general and orderly manner.

That portion of the file system storage complex which is immediately accessible to the file system, i.e. disks and drums, is called the *on-line storage system*. Devices which are removable from the storage complex, such as tapes, data cells and disk packs which are used by the file system as an extension of the on-line facilities, are called the *file backup storage system*. To the user, all files appear to be on line, although access to some files may be somewhat delayed. For the purpose of discussion, a backup system consisting only of magnetic tape is considered. However, the system presented here is readily adaptable to other devices.

Incremental Dumping of New Files

Whenever a user signs off, additional copies of all files created or modified by that user are made in duplicate on a pair of magnetic tapes. At the end of every N hour period, any newly created or modified files which have not previously been dumped are also copied to these tapes. When this is done, the tapes are removed from the machine and replaced by a fresh set of tapes for the next N hour period. Typically N would be a period of between 2 and 4 hours. This procedure has the advantage that the effects of the most catastrophic machine or system failure can be confined to the N hour dumping period.

Weekly Dumping of Frequently Used Files

In the event of a catastrophe, the on-line storage system could be reloaded from these incremental dump tapes. However, since many valuable files, including system programs, may not have been modified for a year or over, this method of reloading is far too impractical. In order to minimize the time necessary to recover after a catastrophe, a weekly dump is prepared of all files which have been used within the last M weeks. This dump is also made on duplicate tapes for reliability.

Actually this weekly dump is taken in two parts. The first part consists of all files which must be present in order to start and run the basic system. The second part consists of all other files which have been used within the last M week period. Typically M would be a period of about three to five weeks. The weekly dump tapes may be released for other use after a period of about two or three months. The incremental dump tapes must be kept indefinitely. However, it may be advantageous to consolidate these tapes periodically by deleting obsolete files.

Catastrophe Reload Procedure

Should a catastrophe occur in which the entire contents of the on-line storage system is lost, the following reload procedure can be used. First reload a copy of the system files from the most recent weekly dump tapes. When this has been done the system may be started, with the rest of the reloading process continuing under the control of the system. Note that this does not necessarily represent the most recent copy of the system. If an important system change has been made since the weekly dump was taken, it may be necessary to reload the incremental tapes before starting the system.

After the system files are reloaded, the incremental dump tapes are reloaded in reverse chronological order, starting with the most recent set of incremental tapes. This process is continued until the time of the last weekly dump is reached. At this time, the second part of the weekly dump tapes is reloaded.

During this process all redundant or obsolete files are ignored. The date and time a file is created or last modified is used to insure that only the most recent copy of a file remains in the on-line storage system. Since directories are dumped and reloaded in the same manner as ordinary files, the contents of the on-line storage system can be accurately restored.

It is possible to continue to load the older weekly tapes until the on-line system is totally reloaded. However, the amount of new information picked up from these tapes becomes increasingly small as one goes further back in time. In view of this, files which do not appear on the most recent set of weekly dump tapes, due to inactivity, are not reloaded at this time. Instead, a trap is added to the appropriate directory branch so that a retrieval procedure is called when the file is first referenced. This allows these files to be reloaded as needed by the retrieval mechanism which is discussed later in this paper.

On-Line Storage Salvage Procedure

Although the catastrophe reload procedure can accurately reconstruct the contents of the on-line storage system, it is normally used only as a backstop against the most catastrophic of machine or system failures. When the milder and more common failures occur, it is often possible to salvage the contents of secondary storage without having to resort to the reload procedure. If this can be done, many files which have been created or modified since the end of the last incremental dump period can be saved. In addition, much of the time necessary to run the reload procedure can also be saved.

The usual result of a machine or system failure is that the contents of secondary storage are left in a state which is inconsistent. For example, two completely unrelated directory entries may end up pointing to the same physical location in secondary storage, while the storage assignment tables indicate that this area of storage is unused. If the system were restarted at this time, the situation might never be resolved. The usual effect is that any information subsequently assigned to that area of secondary storage is likely to be overwritten.

This situation arises when the system goes down before the file system has updated its assignment tables and directories on secondary storage. What has probably happened is that some user has deleted a file and another user created a new file which was assigned to the area of storage just vacated by the previous file. When the system goes down, the changes have not been recorded in secondary storage. This is only one example of the type of trouble which occurs when the system fails unexpectedly.

The salvage procedure is designed to read through all the directories in the hierarchy and correct inconsistent information wherever possible. The remaining erroneous files and directory entries are deleted or truncated at the point at which the error was found. Storage assignment tables are corrected so that only one branch points to the same area of secondary storage. Since it is necessary to read only the directories and the storage assignment tables, the salvage procedure can be run in a small percentage of the time necessary to run a complete reload procedure.

The salvage procedure also serves as a useful diagnostic tool, since it provides a printout of every error found and the action taken. This program can also be run in a mode in which it only detects errors but does not try to correct them.

Retrieval of Files from Backup Storage

Unless a file has been explicitly deleted by a user, the directory entry for that file remains in the file system indefinitely. If, for some reason, the file associated with this entry does not currently reside on an on-line storage device, the corresponding branch for that file contains a trap to a file retrieval procedure. When a user references a file which is in this condition, his process traps to the retrieval procedure. At this time the user may elect to wait until the file is retrieved from the backup system, to request that the file be retrieved while he works on something else, to abort the process that requested the file, or to delete the directory entry.

If the user elects to retrieve the file, the date and time the file was created or last modified (which are available from the directory entry) are used to select the correct set of incremental dump tapes. The retrieval procedure requests the tape operator to find and mount these tapes. These tapes are then searched until the precise copy of the requested file is found and reloaded. At this time the original access control list of the branch is restored, and the file is now ready to be used by the user.

If a user deletes a file, both the file and the corresponding directory entry are deleted. However, if

a copy of this file appears on a set of incremental dump tapes, this copy is not deleted at this time. This file can still be retrieved if the user specifies the approximate date and time when the file was created or last modified. To help the user in this situation, the incremental dump procedure provides a listing for the operations staff of the contents of each set of incremental tapes. These listings are kept in a log book which may be consulted by the operators in situations such as the above. Selected portions of this listing may be made available to the user.

The user is able to declare that he wishes a certain file to be removed from the on-line storage system without deleting the corresponding directory entry. This may be accomplished by using a system procedure which places the file in a state where it can be retrieved by the normal retrieval procedure.

General Reliability

Since the file system is designed to provide the principal information storage facility for all users of the system, the full responsibility for all considerations of reliability rests with the file system. For this reason all dumping, retrieval and reloading procedures use duplicate sets of tapes. These tapes are formatted in such a manner as to minimize the possibility of unrecoverable error conditions. When reading from these tapes during a reload or retrieval process, multiple errors on both sets of tapes can be corrected as long as the errors do not occur in the same physical record of both tapes. If an error occurs which cannot be corrected, only the information which was in error is lost. If the error is a simple parity error, the information is accepted as if no error occurred. When a user first attempts to use a file in which a parity or other error was found, he is notified of this condition through a system procedure using the trap mechanism.

Secondary Storage Allotments

The file system assigns all secondary storage dynamically as needed. In general, no areas of the on-line storage system are permanently assigned to a user. A user may keep an essentially infinite amount of information within the file system. However, it is necessary to control the amount of information which can be kept in the on-line storage system at a time.

When a user first signs on, the file system is given an account name or number. All files subsequently created by this user are labelled with his account name. When the user wishes to increase his usage of secondary storage, the file system calls upon a secondary storage accounting procedure giving the user's account name and the amount and class of storage requested.

The accounting procedure maintains records of all secondary storage usage and allotments. A storage allotment is defined as the amount of information which a particular account is allowed to keep in the on-line storage system at one time. Normally the accounting procedure allows a process to exceed the allotment after informing the file system that the account is overdrawn. However, the accounting procedure may decide to interrupt the user's process if the amount of on-line storage already used seems unreasonable.

Multilevel Nature of Secondary Storage

In most cases a user does not need to know how or where a file is stored by the file system. A user's primary concern is that the file be readily available to him when he needs it. In general, only the file system knows on which device a file resides.

The file system is designed to accommodate any configuration of secondary storage devices. These devices may cover a wide range of speeds and capacities. All considerations of speed and efficiency of storage devices are left to the file system. Thus all user programs and all other system programs are independent of the particular configuration of secondary storage.

All permanent secondary storage devices are assigned a level number according to the relative speed of the device. The devices which have the highest transmission and access rates are assigned the highest level numbers. As files become active, they are automatically moved to the highest-level storage device available. This process is tempered by considerations such as the size of the file and the frequency of use.

As more space is needed on a particular storage device, the least active files are moved to a lower-level storage device. Files which belong to overdrawn accounts are moved first. Files continue to be moved to lower-level storage until the desired amount of higher-level storage is freed. If a file must be moved from the lowest-level on-line

storage device, the file is removed and the branch for this file is set to trap to the retrieval procedure.

4. FILE SYSTEM PROGRAM STRUCTURE

This section describes the basic program structure of the file system presented in the preceding sections, as implemented in the Multics system.[1] (It is assumed here that the reader is familiar with the papers referred to in references 1, 2 and 3.)

A user may reference data elements in a file explicitly through read and write statements, or implicitly by means of segment addressing. It should be noted here that the word "file" is not being used in the traditional sense (i.e., to specify any input or output device). In the Multics system a file is a linear array of data which is referenced by means of a symbolic name or segment number and a linear index. In general, a user will not know how or on what device a file is stored.

A Multics file is a segment, and all segments are files.[1,3] Although a file may sometimes be referenced as an input or an output device, only a file can be referenced through segment addressing. For example, a tape or a teletype cannot be referenced as a segment, and therefore cannot be regarded as a file by this definition.

Input or output requests which are directed to I/O devices other than files (i.e. tapes, teletypes, printers, card readers, etc.) will be processed directly by a Device Interface Module (see reference 4) which is designed to handle I/O requests for that device. However, I/O requests which are directed to a file will be processed by a special procedure known as the File System Interface Module (see reference 4). This module acts as a device interface module for files within the file system. Unlike other device interface modules, this procedure does not explicitly issue I/O requests. Instead, the file system interface module accomplishes its I/O implicitly by means of segment addressing and by issuing declarative calls to the basic file system indicating how certain areas of a segment are to be overlayed.

4.1 The Basic File System

Whether a user refers to a file through the use of read and write statements or by means of segment addressing, ultimately a segment must be made available to his process. The basic file system may now be defined as that part of the central software which manages segments. In general this package performs the following basic functions.

1. Maintain directories of existing segments (files).
2. Make segments available to a process upon request.
3. Create new segments.
4. Delete existing segments.

Figure 3 is a rough block diagram of the modules which make up the basic file system. This diagram is by no means complete but is used here to give the reader an overall view of the basic flow. The directional lines indicate the flow of control through the use of formal calling sequences, with formal return implied. Lines with double arrowheads are used to indicate possible flow of control in either direction. The circles in the diagram indicate some of the data bases which are common to the modules indicated. The modules and data bases drawn below the dotted line must at least partially reside in core memory at all times since they will be invoked during a missing-page fault (see reference 3).

Segment Management Module

The segment management module maintains records of all segments which are known to the current process. A segment is *known to a process* once a segment number has been assigned to that segment for this process. A segment which is known to a process is *active* if the page table for that segment is currently in core. If the page table is not currently in core, that segment is *inactive*.

If a segment is known to a process, an entry will exist for that segment in the Segment Name Table (SNT). This entry contains the call name, the tree name and the segment number of the segment (file) along with other information pertinent to the segment as used by this process. The *call name* is a symbolic name used by the user to reference a segment. This name normally corresponds to an entry in the user's directory hierarchy which effectively points to the desired file. It should be noted that a different copy of the segment name table exists for each individual process.

If a segment is active, an entry for that segment exists in the Segment Status Table (SST). This ta-

ble is common to all processes and contains an entry for each active segment. If a segment is inactive (no page table is in core), no entry exists for that segment in this table. Each entry in the segment status table contains information such as the number of processes to which this segment is known and a pointer which may be used to reference the file or files which are to receive all I/O resulting from paging this segment in and out of core.[3]

When a user references a segment for the first time, a directed fault will occur. At this time control is passed to a procedure known as the linker.[3] This procedure picks up the symbolic segment call name from a pointer contained in the machine word causing the fault. The linker must now establish a segment number from this symbolic name. An entry to the segment management module is provided for precisely this purpose.

When a call is made to the segment management module to establish a segment number from a call name, the segment name table is searched for that call name. If the call name is found in the segment name table, the segment number from this table is returned immediately to the calling procedure. However, if this is not the case, the segment management module must take the following steps.

1. Locate the segment (file) in the user's directory hierarchy via a call to the search module.
2. Assign a segment number for this segment.
3. Update the segment name table indicating that this segment is now known to this process.
4. Open the file or files which are to receive I/O resulting from paging.
5. Create or update the appropriate entry in the segment status table.
6. Establish a page table and segment descriptor for this segment if the segment was not already active for some other process.
7. Return the segment number to the calling procedure.

If a segment is known to a process but is not currently active, the descriptor for that segment will indicate a fault condition. If and when this fault occurs, the segment can be reactivated by locating the appropriate entry in the segment name table and repeating steps 4 through 7. Note that the segment does not have to be located again in the directory hierarchy since the tree name is retained in the segment name table.

If a segment is to be modified during its use in a process, the user may elect to modify a copy of that segment rather than the original. When this is the case, the copying of this segment is done dynamically as a by-product of paging. However, if the copying is not complete at the time the segment becomes inactive, the copying must be completed at this time.

If a segment is to be copied, there are actually two open files involved, the original file and the copy or *execution file*. When a page table is initially constructed by the segment management module, each entry in that page table will contain a fault indication and a flag indicating what action should be taken if and when that fault occurs. This flag may indicate one of the following actions:

1. Assign a blank page.
2. Retrieve the missing page from the original file.
3. Retrieve the missing page from the execution file.

Once a page has been paged out (written) into the execution file, it must be retrieved from that file.

An entry to the segment management module is provided by which a user may declare a synonym or list of synonyms for a segment name. For example, a user may have a certain procedure which references a segment called "Gamma" and another procedure which references a segment called "Alpha." If the user wishes to operate both procedures as part of the same process using a segment called "Data" he may do so by declaring Alpha and Gamma to be synonyms for Data. This association is kept by the segment management module in a Synonym Table (SYNT). Whenever the segment management module is presented with a call name which has been defined as a synonym, the appropriate name is substituted before any further processing takes place.

In addition to the functions described above, the segment management module provides entries through which the user may ask questions or make declarations involving the use of segments known to his process. Some of these functions are listed below.

1. Declare that a segment or some specific locations within a segment are no longer needed at this time.
2. Declare that a segment or some specific locations within a segment are to be reassigned rather than paged in as needed. (The user is about to overwrite these locations).
3. Ask if a segment or some specific locations within a segment are currently in core.
4. Declare that a certain segment is to be created when first referenced.
5. Terminate a segment, indicating that this segment is no longer to be considered as known to this process.

Search Module

The search module is called by the segment management module to find a particular segment (file) in the user's directory hierarchy. The search module directs the search of individual directories in the user's hierarchy in a predetermined pattern until the requested branch is found or the algorithm is exhausted. This module calls the file coordinator to search particular directories and to move to other directories in the hierarchy. The user is able to override this search procedure by providing his own search procedure at the initiation or during the execution of his process.

The File Coordinator

The file coordinator provides all the basic tools for manipulating entries within the user's current working directory. The functions provided by this module perform only the most primitive operations and are usually augmented by more elaborate system library procedures. The following is a list of some of these operations.

1. Create a new directory entry.
2. Delete an existing entry.
3. Rename an entry.
4. Return status information concerning a particular entry.
5. Change the access control list for a particular branch.
6. Change working directory.

Whenever a user wishes to perform any operation through the use of the file coordinator, the access control module is consulted to determine if the operation is to be permitted.

Since most calls to the file coordinator refer to entries contained in the user's working directory, the file coordinator must maintain a pointer to this directory. This is done by keeping the tree name of the working directory in a Working Directory Table (WDT) for this process.

Directory Management Module

When the file coordinator wishes to search the user's working directory, the actual search is accomplished by use of the directory management module. This module searches a single directory specified by a tree name for a particular entry or group of entries. The actual directory search is confined to this module to isolate the recursion process which may be required to search a given directory.

The directory management module issues calls to the segment management module to obtain a segment number for the directory for which it has only a tree name. When the directory management module obtains this segment number and references the directory by means of segment addressing, a descriptor fault may occur indicating that this segment is no longer active. If this happens, the segment management module will try to reactivate this segment by attempting to find this directory in the next superior directory by means of the tree name in the segment name table. To do this the segment management module issues a direct call to the directory management module to search the next superior directory for the missing directory. After obtaining a segment number for the superior directory, the directory management module may cause another descriptor fault to occur when attempting to search this directory. This process may continue until a directory is found to be an active segment or until the root of the directory hierarchy is reached. Since the root is always known to the directory management module, the depth of recursion is finite.

File Control Module

The file control module is provided to open and close files for the segment management module. A

file is said to be open, or *active*, if it has a corresponding entry in the Active File Table (AFT). If a file is active, the corresponding entry in the active file table provides sufficient information to control subsequent I/O requests for that file.

If the file is inactive, the open procedure needs only to open the file to the requested state and make the corresponding entry in the active file table. If the file is active, it may have N users reading, or 1 user reading and writing, or N users data sharing (using file as a common data base). If the requested state is incompatible with the current state of the file, the current process must be blocked.[3] For example, if the current user wishes to read a stable copy of the file and there is currently a user writing into that file, the requested state (reading) and the current state (reading and writing) are said to be incompatible.

If the requested state and the current state of the file are found to be compatible, the number of users using the file in that state is increased by one. When a file has been successfully opened by the file control module (with the permission of the access control module), the pointer to the corresponding entry in the active file table is returned to the calling procedure. This pointer is used to direct requests for subsequent input or output to the correct file.

Access Control Module

The access control module is called to evaluate the access control information for a particular branch, as defined in Section 2. This module is given a pointer to the directory entry for the branch in question and a code indicating the type of operation which is being attempted. The access control module returns a single effective mode to the calling procedure. The effective mode is the mode which governs the use of a file with respect to the current user or process. The calling procedure uses this mode to determine if the requested operation is to be permitted.

If the access control information indicates that a trap is to be effected, the procedure to which the trap is directed is passed the entry for the branch in question and the operation code. The procedure which processes the trap must return to the access control module, specifying the effective mode to be returned by the access control module to its calling procedure. The procedure which processes the trap may choose to strengthen, weaken or leave unchanged the usage attributes which define the effective mode for the branch.

Page Marker Module

The page marker periodically interrupts the current process and takes note of page usage, and resets the page use bits[2] of all pages involved in the current process. Pages which fall below a dynamically set activity threshold are listed in the Page Out Table (POT) as likely candidates for removal when space becomes needed.

Page Management Module

Control passes to the page management module by means of a missing-page fault in a page table in use by the current process. This fault may indicate that a new page should be assigned from free storage or that an existing page should be retrieved from an active file. In either case a free page must be assigned before anything else can happen. If no pages are currently available, the first page listed in the page out table is paged out. If no pages are listed in the page out table, a random page of appropriate size is removed.

If a new page is to be read in, the page table entry for the missing page contains a pointer to the appropriate entry in the segment status table and a flag indicating whether this page is to be read from the original file or the execution file. In either case a pointer to the appropriate active file may be obtained from the segment status table. This pointer is passed as a parameter to the I/O queue management module with a read request to restore the correct page to core memory.

I/O Queue Management Module

The I/O queue management module processes input and output requests for a particular active file. The calling procedure specifies a read or a write request and a pointer to an entry in the active file table which corresponds to the desired file. This request is placed on the appropriate queue for the particular device interface module which will process the request. The queue management module then calls that device interface module indicating that a new request has been placed on its queue. When this is done, the queue management module

returns to the calling procedure which must decide whether or not to block itself until the I/O request or requests are completed.

Device Interface Modules

For each type of secondary storage device used by the basic file system, a device interface module will be provided. A device interface module has the sole responsibility for the strategy to be used in dealing with the particular device for which it was written. Any special considerations pertaining to a particular storage device are invisible to all modules except the interface module for that device.

A device interface module is also responsible for assigning physical storage areas, as needed, on the device for which it was written. To accomplish this function, the interface module must maintain records of all storage already assigned on that device. These records are kept in *storage assignment tables* which reside on the device to which they refer.

4.2 Other File System Modules

The modules described below are not considered part of the basic file system and are not indicated in Fig. 3. However, these modules are considered to be a necessary and integral part of the file system as a whole.

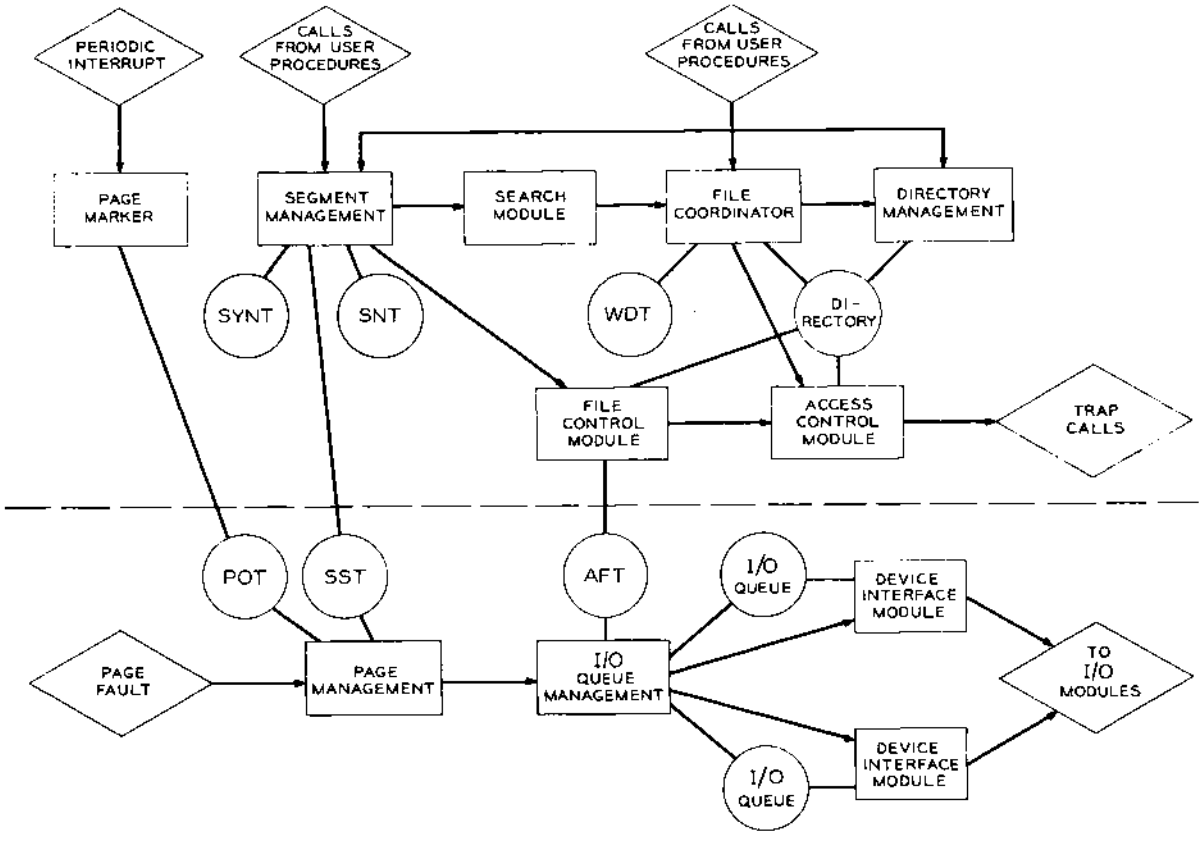

Figure 3. The basic file system.

Multilevel Storage Management Module

The multilevel storage management module operates as an independent process within the Multics system. This module collects information concerning the frequency of use of files currently active in the system. In addition, this module collects information concerning overdrawn accounts from the secondary storage accounting module.

The storage management module insures that an adequate amount of secondary storage is available to the basic file system at all times. This is accomplished by moving infrequently used files downward in the multilevel storage complex. This module also moves the most frequently used files to the highest-level secondary storage device available.

Storage Backup System Modules

The storage backup system consists of five modules which operate as independent processes. These modules perform the functions described in Section 3.

1. Incremental Dump Module—The sole responsibility of this module is to prepare incremental dump tapes of all new or recently modified files.
2. Weekly Dump Module—This module is run once a week to prepare the weekly dump tapes.
3. Retrieval Module—This module retrieves files which have been removed from the on-line storage system.
4. Salvage Module—This module is run after a machine or system failure to correct any inconsistencies which may have resulted in the on-line storage system. Since the Multics system cannot safely be run until these inconsistencies are corrected, the salvage module must be capable of running on a raw machine.
5. Catastrophe Reload Module—This module is used to reload the contents of the on-line storage system from the incremental and weekly dump tapes after a machine or system failure. Normally, this module is run only when all attempts to salvage the contents of the on-line storage system have failed. This module must be capable of running on a raw machine or under the control of the Multics system.

Utility and Service Modules

A large library of utility modules is provided as part of the file system. These modules provide all the necessary functions for manipulating links, and branches using the more primitive functions provided by the file coordinator.

A special group of utility modules is provided to copy information currently stored as a file to other input or output media, and vice versa. The following functions are provided as a bare minimum:

1. File to printer
2. File to cards
3. Cards to file
4. Tape to file
5. File to tape

Actually these modules merely place the user's request on a queue for subsequent processing by the appropriate service module. The service module executes the requests in its queue as an independent process. As soon as the user's request has been placed on an appropriate queue, control is returned to the calling procedure although the request has not yet been executed.

5. CONCLUSIONS

In this paper, a versatile secondary storage file system is presented. Various goals which such a system should attain have been set, and the system designed in such a way as to achieve these goals. Such a system is felt to be an essential part of an effective on-line interactive computing system.

6. ACKNOWLEDGMENT

The file system presented here is the result of a series of contributions by numerous people, beginning with the MIT Computation Center, continuing with Project MAC, and culminating in the present effort.

REFERENCES

1. F. J. Corbató and V. A. Vyssotsky, "Introduction and Overview of the Multics System," this volume.

2. E. L. Glaser, J. F. Couleur and G. A. Oliver, "System Design of a Computer for Time-Sharing Applications," this volume.

3. V. A. Vyssotsky, F. J. Corbató and R. M. Graham, "Structure of the Multics Supervisor," this volume.

4. J. F. Ossanna, L. E. Mikus and S. D. Dunten, "Communications and Input-Output Switching in a Multiplex Computing System," this volume.

5. E. E. David, Jr., and R. M. Fano, "Some Thoughts About the Social Implications of Accessible Computing," this volume.

Additional References

C. W. Bachman and S. B. Williams, "A General Purpose Programming System for Random Access Memories," Proceedings of the Fall Joint Computer Conference 26, Spartan Books, Baltimore, 1964.

J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *ACM Conference on Programming Languages,* San Dimas, Calif., Aug. 1965. To be published in *Comm. ACM.*

A. W. Holt, "Program Organization and Record Keeping for Dynamic Storage Allocation," *Comm. ACM* 4, pp. 422-431, Oct. 1961.

T. H. Nelson, "A File Structure for the Complex, the Changing and the Indeterminate," *ACM National Conference,* Aug. 1965.

M. V. Wilkes, "A Programmer's Utility Filing System," *Computer Journal* 7, pp. 180-184, Oct.-1964.

COMMUNICATIONS AND INPUT/OUTPUT SWITCHING IN A MULTIPLEX COMPUTING SYSTEM*

J. F. Ossanna
Bell Telephone Laboratories
Murray Hill, New Jersey

L. E. Mikus
General Electric Company
Phoenix, Arizona

S. D. Dunten
Massachusetts Institute of Technology
Cambridge, Massachusetts

INTRODUCTION

This paper discusses the general communications and input/output switching problems in a large-scale multiplexed computing system. A basic goal of such a computing system is to serve simultaneously and continuously a wide range and large number of users. By rapidly time-multiplexing the use of computer system facilities on behalf of these users, the system attempts to satisfy the completion time and response time desires of both the on-line interactive user and the absentee user.

Problems arise in such systems because of the large number and variety of on-line input/output devices, the dynamically changing hardware and software environment, and the need to efficiently use devices such as line printers.

In this paper a new general purpose input/output controller is described which is capable of simultaneously operating a large number of devices of almost arbitrary variety and speed. An input/output software system philosophy is presented which is tailored to the environment of a multiplex computer system. It includes a message coordinator which connects a user program's input and output streams to various input/output devices and to the secondary storage file system. Execution-time redirection and multiple-direction of these connections is a software system feature.

This paper represents the philosophy and direction of the development of the communications and input/output portions of the Multics system (*Multi*plexed *I*nformation and *C*omputing *S*ystem).[1-5]

GENERAL PROBLEMS

The on-line input/output devices in a computer system may be classified as local or remote. The local peripherals such as drums, discs, tapes, line printers, card readers and punches are typically con-

*Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-102(01).

nected to the computer by short, many-conductor cables. The computer system usually has considerable status information available about local devices and has operators to assist in their care and feeding.

By comparison, remote devices such as typewriters, line printers, card readers and punches are typically connected to the computer system via private or switched telephone company transmission facilities. The computer system can directly obtain only some status of the transmission facilities. The remote operator, if any, is likely to be reachable only via his remote terminal. The number of remote terminals will generally be large compared to the number of local peripherals, and will vary dynamically with the number of remote users.

The actual and potential variety of input/output devices is a challenge to a large-scale multiplex computer system. Likely additions to the devices mentioned above are removable discs, some form of inexpensive mass storage such as the data cell, a variety of both local and remote graphical display terminals, a microfilm processor, remote data collectors and various dependent peripheral analog and digital computers. Further, devices like remote typewriters and line printers will typically abound in several models. In some cases the computer itself will need to originate calls to remote terminals.

Certain remote terminals can impose stringent real-time response obligations on a multiplex computer system. Examples are remote process control or experiment control and certain types of discontinuous remote high-speed data collection.

Supervisory program modules which attend to bulk input/output devices such as line printers and card readers must be scheduled for execution frequently enough to guarantee the efficient use of these devices. Because the supervisor is multiprogramming (processing in parallel) the programs of perhaps hundreds of users, the problem of allocating such facilities as tapes and removable discs to these user programs is nontrivial.

INPUT/OUTPUT HARDWARE SYSTEM PHILOSOPHY

A basic goal of the Multics system is continuous service. The principal method in achieving this goal is to include more than one copy of each hardware module and to configure the connecting paths between modules such that no single module is essential for continued system operation.

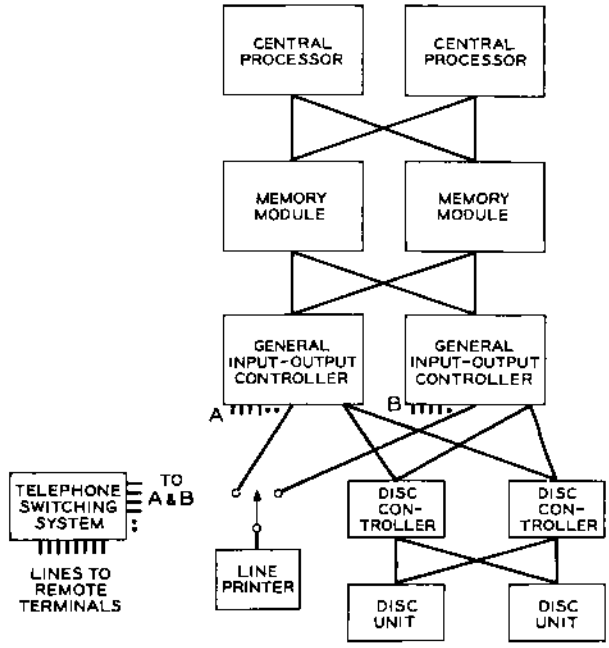

Figure 1. Skeletal hardware system configuration.

Figure 1 is a skeletal system configuration, and shows just enough modules to illustrate this principle. In this example, there are two central processors (CPU), two memory modules, two general input/output controllers (GIOC), two disc controllers, and two disc units. In each case the number two is illustrative and can be higher. Each CPU and GIOC can access every memory module. The disc controller is twin-tailed and accessible by either GIOC; each disc is reachable from either disc controller. Thus there are multiple paths for data flow between disc and main memory.

Single-tailed input/output devices (connectible to only one controller at a time), illustrated by the line printer in Fig. 1, must be manually switched when necessary by the local computer operator.

Remote terminals such as typewriters may access the system via some switching system such as a regular telephone central office or private branch exchange. A number of lines connect each GIOC to the switching system providing multiple-path availability. GIOC ports intended for different types of remote terminals operating at different transmission rates can be assigned different telephone numbers. The switching of remote terminals is discussed below (Connecting and Switching Remote Terminals).

A GENERAL INPUT/OUTPUT CONTROLLER

The need to handle simultaneously input/output devices having a wide range of speeds, and the need to impose automatically priorities dictated not only by this speed range but also by real-time requirements and by the actual relative importance of different terminals, have motivated the design of a generalized input/output controller (GIOC). The GIOC, conceived for the real-time environment, achieves several functions not highly developed in current large-scale computer input/output subsystems.

A tradeoff between hardware complexity and memory usage is available for all speed ranges of terminal devices, whether they be 10-character-per-second teletypewriters or 400,000-character-per second mass storage peripherals. A modular organization allows functional building blocks to be assembled and tailored specifically for the terminal complement of any rational system.

The input/output capacity of the system is increased over present computer systems by a hardware priority scheme which considers individually the allowable latency of every event requiring the use of main memory. No event requiring the use of main memory is given any higher priority than it requires for error-free operation.

Corresponding to the hardware priority scheme for memory usage is a hierarchy of program interrupt priorities that can ensure rapid response times for real-time events at the expense of slower response times for nonreal-time events.

To facilitate real-time responses to terminal devices, channel commands may be executed without program intervention. By placing commands in a list of channel control words, the commands can be conditionally or unconditionally triggered by the data stream.

All input/output operations are under the direct control of an input/output software system, and hardware memory protection for input/output is not required. Uniform programming is facilitated by the implementation of identical formats and procedures for the control of all terminal devices.

Functional Division

The functional division of the GIOC hardware is illustrated in Fig. 2. The modular functional building blocks are the common control, adapter control, and adapter channels.

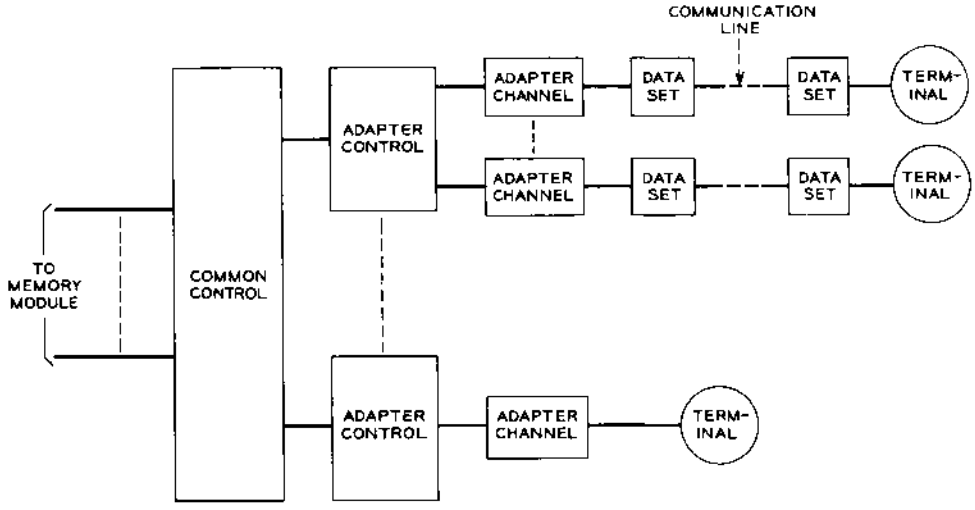

Figure 2. Functional organization of the input/output controller.

Two general types of adapters are used—direct and indirect. Direct adapters, for which the active control word for data transfer resides in the adapter hardware rather than in main memory, are employed with high-data-rate devices. Data are transferred directly to or from the designated locations in main memory using the control word resident in the adapter hardware. By transferring up to 12 characters in a single main memory cycle, direct adapters efficiently utilize memory cycles at the expense of adapter complexity.

Indirect adapters, which contain minimum control within themselves, are employed with low- and medium-data-rate devices. Control words reside in main memory and must be accessed and updated every time a data transfer occurs. Up to 12 charac-

ters of data can be transferred in three main memory cycles.

Adapter channels supply the proper termination to lines connecting the terminal devices to the GIOC. Adapter channels may contain data buffering or simply supply the line interface.

The common control section does not contain any data buffering, but serves mainly to order and control all data transfers. For direct adapters, the common control provides the functions of:

1. Adapter priority ordering and allocation.
2. Communicating program command information.
3. Interfacing to the adapters.
4. Sequencing and subsystem control.
5. Diagnostics and error control.

In addition to the above, for indirect adapters the common control also provides the functions of:

6. Word assembly and disassembly.
7. Parity checking and generation.
8. Control word updating.
9. Dynamic control function detection.
10. Temporary adapter status buffering.

For control of data transfers, the mailbox technique is used. Each adapter channel has a dedicated area, or mailbox, in protected main memory for the residence of control words. Data transfer control words are independent for every adapter channel but command, diagnostic, and status control words are shared by all adapter channels.

Each adapter channel mailbox contains two types of control words: Data Control Words (DCWs) and List Pointer Words (LPWs). DCWs control the data transfer between main memory and adapter channels. Each adapter channel has associated with it an LPW which is a pointer for locating the next DCW to be used for data transfer.

Except for command initiation and termination of unit record peripherals, all information necessary to carry out an input/output operation is uniquely contained within the control words of each adapter channel. In this manner, no adapter channel is dependent upon any other adapter channel in the GIOC for control.

Hardware Priority Scheme

A hardware priority scheme minimizes the effects of low-data-rate devices upon the latency of high-data-rate devices by the establishment of priority for all events which require the use of main memory. Six general classes of events exist:

1. Status (communication from the hardware to the program).
2. Commands (communication from the program to the hardware).
3. Direct adapter data services.
4. Indirect adapter data services.
5. List pointer services.
6. Diagnostic functions.

Each class of events itself contains several levels of hardware priority. The priority levels for one class of events can be intermixed with those of another. For example, seven levels of normal status priority are allowed. These seven levels may be intermixed with the priority levels of the other classes.

Commands are treated in the same way as all other events which require the use of main memory. They must wait until the common control grants priority for their issuance.

Two levels of hardware priority for commands are allowed. To issue commands, the program loads one of the two allocated mailboxes with a pointer to the command list. The program then issues an interrupt to the common control. One command is executed each time this event receives priority, until no further commands remain in the command list.

Indirect data service events are those required to transfer data between main memory and indirect adapter channels. Each data transfer takes three main memory cycles.

Direct data service events are those required to transfer data between main memory and direct adapter channels. Each data transfer takes one main memory cycle.

Whenever a DCW exhausts, a list pointer service is required to obtain another DCW and place the new DCW in the proper mailbox. An LPW contains the address of the next DCW to be used in scatter-gather operations.

Instead of accessing the LPW immediately upon a DCW exhaust, the list pointer service event is given a priority just higher than the data service priority for that adapter channel. To initiate the list pointer service, priority must first be granted for that list pointer service event. In this way, the new DCW is guaranteed to be in the proper mailbox before the next data service for that adapter channel occurs. However, the list pointer service time does

not add to the latency of higher priority events. This feature allows low-speed terminal devices to operate using sophisticated scatter-gather and control techniques without adding additional latency to higher priority events.

All work to be done on the GIOC is partitioned into events in such a way that no single event requires more than four accesses to main memory. With the exception of direct adapter data services, priority is allocated to a new event at the completion of every current event.

To further reduce latency for high-transfer-rate terminal devices, any direct adapter data service can temporarily preempt the priority of any lower priority event and thus gain access for the next main memory cycle. Thus, high priority direct adapter data services do not wait for other events using multiple memory cycles (such as indirect adapter data service) to complete their sequence before gaining access to main memory.

Each adapter uses one or more levels of hardware priority. All adapter channels within a single adapter are assigned subpriorities among themselves by the adapter control.

A complete look at all levels of hardware priority is taken after the completion of every event, and the event which will receive the next memory access is then determined. This priority determination occurs concurrently with other common control functions. In effect, after every event the priorities of all events requiring further memory accesses are reconsidered, and the allocating of the next event to receive a memory access is granted.

Except for direct channel data services, which can temporarily preempt the priority of other events, any event which requires the services of main memory is guaranteed to be recognized in its allocated hardware priority sequence within four main-memory access times used by the common control.

Program Interrupt Priorities

Program interrupt priorities can be dynamically assigned by the supervisor on a per-channel basis. Program interrupts result from status being stored in main memory after an event occurs which requires program action.

In addition to the seven priority levels for memory access provided by the hardware, status has seven levels of program interrupt priority. For any given event, when a level of hardware status priority is changed by the software, its corresponding level of program interrupt priority is also automatically changed, guaranteeing the associated change in real-time response for that event.

Four subclasses of status events exist:

1. Exhaust
2. Terminate
3. External signal
4. Internal signal

Exhaust status indicates the current active control word for an adapter channel cannot be used for further control. This event implies that a new control word must be obtained to continue data transfer.

Terminate status indicates that the current active control word for an adapter channel cannot be used for further control and that, in addition, no further data transfer for that adapter channel is allowed.

The external signal status is the vehicle by which events outside the GIOC can gain recognition by the program. Such events as operator actions on peripherals fall into this subclass.

The internal signal status is the vehicle by which special control events within the GIOC can gain recognition by the program. Such events as the dynamic detection of incoming communication control characters fall into this subclass.

Each adapter channel, through its control words, can independently activate any one of the status levels when one of the four subclasses of status events occurs. For any given adapter channel, the status levels associated with the subclasses of events may be the same or different. At any given time, each adapter channel thus has access to four levels of status response corresponding to the four subclasses of status events.

Under program control, the same status event occurring on different adapter channels can be assigned different levels of hardware priority and correspondingly different levels of program interrupt priority. This allows optimization of the real-time effect of any event upon any other queued events.

Channel Commands

Channel commands can be accepted by an adapter channel at any time, but not all allowable commands have rational meaning when executed without program intervention. Command execution

without program intervention can occur only as a result of a data transfer.

Commands such as "change from transmit to receive mode" can be preplanned in the data sequence and executed without the need for program cognizance at the time of execution. Other commands, such as "change from the inactive to active mode" only have meaning when initiated by the program.

INPUT/OUTPUT SOFTWARE PHILOSOPHY

The input/output software philosophy must simplify wherever possible the design of a large-scale multiplex computer system and must adequately cope with the general communication and input/output problems discussed earlier.

Modularity

In order to accommodate a dynamically changing device environment and to permit the introduction of new input/output devices without major effort, the input/output software should be highly modular. The device-dependent software for each device should be isolated in a separate, replaceable module. Figure 3 shows a general overall block diagram of the input/output software and its relation-

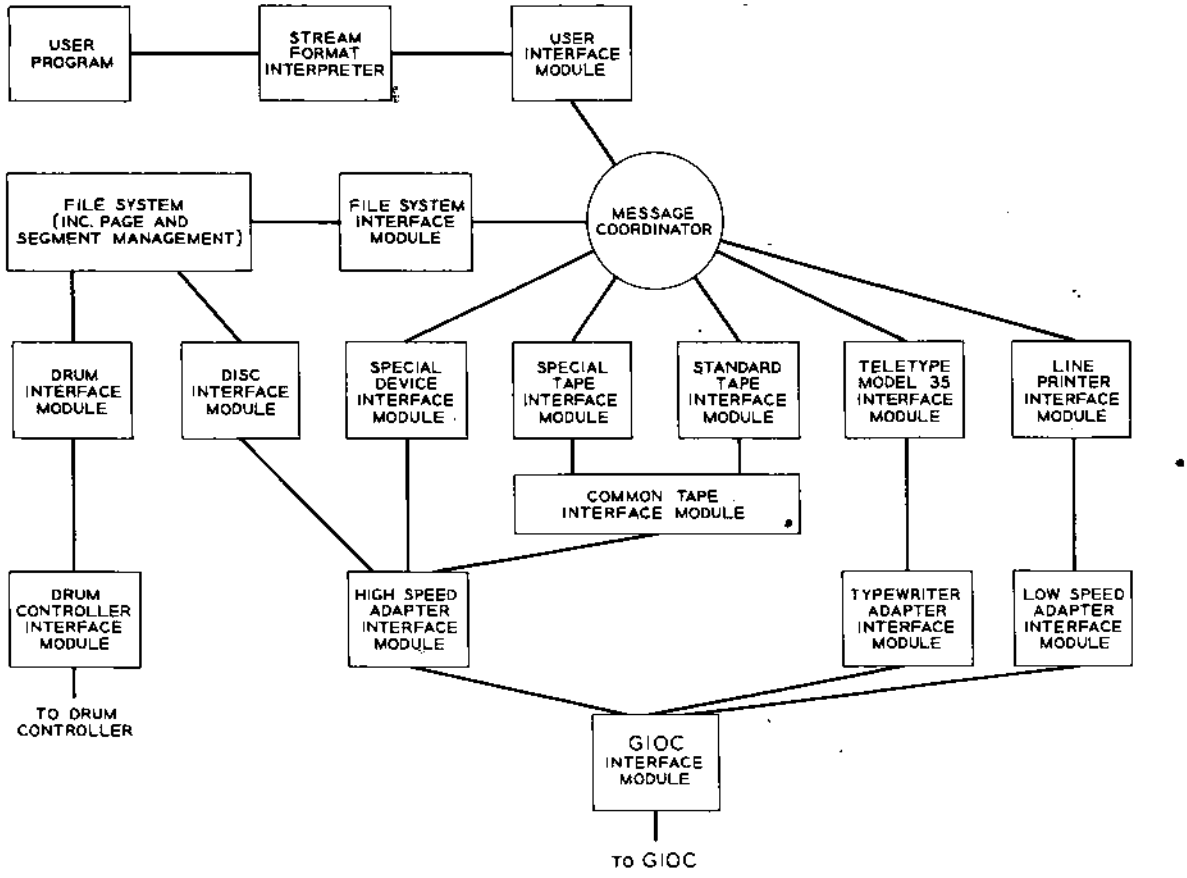

Figure 3. Block diagram of input/output software system.

ship to other software. The message coordinator (MC) and other blocks are discussed below. The MC essentially switches a user's input/output streams to various device interface modules (DIM). A standard DIM will exist for each type of device; for example, there will be a DIM to operate all currently attached Teletype Model 35 typewriters using a standard strategy. This DIM as well as others must be replaceable during system operation. In addition, it must be possible to add a second DIM to operate one or more Model 35's with some special strategy, and to easily associate it with the proper user's streams and the proper communication lines (i.e., the proper Model 35's). This indicates the necessity

for a well-defined, standard software interface for DIMs. The same standard interface must apply to the user interface modules (UIM) and the file system interface module (FSIM).

Because the GIOC is a device, all software which knows about standard GIOC behavior is isolated in the controller interface module (CIM) for the GIOC (the GIOC CIM is hereafter called the GIM). Inasmuch as the GIOC can accommodate various adapter control modules, any special behavior inherent in one kind of adapter control should also be isolated. The typewriter adapter control and adapter channels, for example, may have an echoplex feature (retransmit what is being received), which has no connection with the basic GIOC philosophy or with any particular typewriter. Thus a need exists for adapter interface modules (AIM).

Flexible Input/Output Direction

It should not be necessary for a user to decide at the time he writes a program what actual sources and destinations are to be associated with his program input/output streams. The term "stream" is used here to include all input/output transactions, whether they be sequential access or random access in nature. For example, the "title" of a "file" in a PL/I program is the "stream" name to the input/output system.

Prior to execution, a declaration to the UIM establishes a stream-device connection. The absence of such a declaration implies a default connection; a program invoked from a remote typewriter would have that typewriter connected to all input/output streams in the default case. It must be possible to alter this connection during execution by a call to the UIM. Further, it must be possible to multiple-connect streams and devices. During debugging, a particular program might direct both bulk-production printout and commentary printout to a remote typewriter. Later the bulk printed output might be directed to a computation center high-speed line printer or a file in the secondary storage file system. At another time, some printout may need to be directed simultaneously to two line printers in two different locations and to a tape file.

It should be understood that the "user program" may be some supervisory system module. Printer-destined streams are normally diverted to a file in the file system for later scheduled printing, unless the user really meant to attach a printer to his program. Such diversion is also useful for output on tape, removable disc, etc., to facilitate allocation of such devices.

Description

A user program initiates input/output by calling a standard or special UIM. The call arguments include the stream name. The redirection and multiple-direction of streams and devices require a module which acts as a switchboard; the message coordinator (MC) in Fig. 3 has this purpose. The UIM uses the MC to implement the stream-device connections for a call. If a stream is associated with a file in the File System (FS), a connection is made to the FSIM.

The File System contains (or knows how to retrieve) all retained files; it is described in detail in a companion paper.[4] Files in the File System are essentially formatless. The FSIM can impose a standard file format. The FSIM makes only declarative calls to the File System; it accomplishes the input/output of files implicitly by means of segment addressing.[3,4]

Figure 3 shows a few representative DIMs. The DIMs basically invoke the strategy for handling particular devices. For example, they may convert between the system's standard character set and the device's particular character set. This conversion includes handling of escape conventions necessary to represent characters absent on the device. The Model 35 DIM knows what messages are needed to operate various terminal features. Two standard tape DIMs are shown, because of the need for two distinct tape strategies. One tape DIM handles tapes in standard system format. The second tape DIM permits handling of nonstandard tape formats in a standard way. Both of these tape DIMs call a subsidiary tape DIM which handles common tape problems.

A single level of format interpretation is available from the DIMs and the FSIM. File System files and device data can be interpreted as being formatless or as having arbitrarily long logical records. In the latter case the files and data must contain format information. Any additional format interpretation such as that required by PL/I can be done by the user or by some standard stream format interpreter (see Fig. 3).

Certain DIMs, such as the disk DIM used exclusively by the File System, do not use the Message

Coordinator. Some may not funnel through the GIM; this is illustrated by the drum DIM and separate drum controller CIM in Fig. 3. The File System may also directly call certain DIMs that are normally reached via the message coordinator. For example, disc pacs can be used for extending the File System storage.

An input/output software interface language which is independent of the computer, of the input/output controllers, and of the input/output devices themselves should be used for communicating between software modules. The ease of adding, substituting, and replacing modules implies the need for every module to check the validity of each call to it. For example, the GIM must determine whether a request for service from a DIM or AIM is valid—perhaps whether or not the requester has the right to initiate activity on the referenced channel. The interface language must facilitate this validity checking. All address references in the language are relative. An inner module in the GIM will translate to absolute addresses when actual DCWs are formed.

Interrupt Handling. All basic trap or interrupt handling is begun in a supervisor module outside the input/output system.[3] This module determines which module of which process is to be informed about the interrupt. Interrupts originating from the GIOC, for example, are passed for handling to the GIM which knows how to disentangle the associated status information from the GIOC. In turn the GIM passes back to the Tape DIM interrupt and status information relevant to tape handling. Certain interrupts might ultimately be reflected back to a user process.

Random and sequential input/output calls are permitted to be mixed and used for all of a user's input/output streams. Sequential calls include calls for the next record, message, character, etc., and calls for spacing and backspacing. A call for "record fourteen" is a random call. All DIMs and the FSIM shall take some action for every type of call. A call to backspace the card reader may result in an error return or no-operation depending on circumstances. Backspacing a typewriter with reverse line feed might be valid. Random calls to a tape file are permitted, because of the inclusion of logical record numbers within the logical record on the tape file.

There is no intended direct correlation between the type of call and efficient device utilization. The user of files in the file system will not usually know on what physical device the file exists. Even if the user did, the file may be scattered on the device in an unknown way. The multiplex character of the monitor system will overlap rewinds, seeks, etc.

Synchronous and asynchronous input/output are the two basic operating modes for any particular input/output stream. In the asynchronous mode, the physical input/output transactions are not necessarily synchronized or interlocked with the execution of a program's input/output statements. For example, a user at a typewriter would be allowed to type messages into the system prior to the execution of the read statement which would use them; every execution of a read statement merely plucks the next waiting message out of an input buffer. This example of asynchronous input is analogous to buffered read-ahead schemes which have been used with discs, tapes, etc. An example of asynchronous output is the collecting of output in a core buffer until some physical record size is reached.

In the synchronous mode, the physical transaction associated with a program's input/output statement is carried out during the statement's execution; i.e., control is not returned to the program until the actual transaction is completed. For example, a typewriter user would not be allowed to type (the keyboard might be locked) until the read statement was encountered.

For a particular stream, the input and output modes are independent; for example, the input might be interlocked and the output not. The modes are declarable both prior to and during execution by calls to the UIM. Appropriate interpretation of these modes appears possible for multiple-connected streams and devices. Establishment of a mode amounts to determining which system module in the chain initiates the return to the user program.

Under most circumstances asynchronous input/output is the most efficient. The synchronous or interlocked input/output is useful when operator or user attention is required, and most important when a user is interacting with an undebugged, strange, or many-branched program. The synchronous mode should be imposed on a remote terminal whenever a stream is not associated with the terminal, i.e., when there is no program to which to give messages.

Statistics. Sufficient statistical collecting ability must be included in the input/output software de-

sign to accommodate almost any conceivable charging and facilities-allocation schemes. Modes of operation for taking extensive data relevant to system performance should be possible.

REMOTE TERMINAL CHARACTERISTICS

Remote terminals may be classified as independent or controlled, insofar as the computer system is concerned. A remote small computer which interrupts occasionally for a fast calculation is largely independent. A typical remote typewriter is completely controlled when connected to the system. The following discussion pertains to controlled terminals generally and is illustrated by reference to remote typewriters. The discussion is not intended to provide a list of all of the desirable remote typewriter characteristics.

Status

It is important that the system always have as complete a knowledge of terminal status as possible. Therefore, all pertinent terminal functions must be accompanied by transmission to the system of appropriate information. For example, line feed, carriage return, ribbon color shift, etc., on a typewriter all must transmit characters to the system. Of course, these same functions must be performable by the system by transmission of suitable codes to the terminal. The Proposed Revised ASCII character set provides 32 control characters.

Terminal Lock

To implement the synchronous input mode, the terminal must be lockable by the system. When a read statement is executed, the typewriter keyboard can be unlocked by the system. Even in the asynchronous input mode the keyboard should not be unlocked until the input/output software and hardware are ready to buffer a message. The inability to lock a terminal is an invitation to unexpected and/or unwanted input. The terminal is typically locked during computer output. Of course, a printed, audible, or preferably visual proceed indication is needed to alert the user that input is possible.

An alternative to a terminal lock on terminals producing their own local copy of the input is to operate them full-duplex and to have the computer system echo or retransmit the input back for display. The lack of local copy becomes an indication that input is not wanted. This scheme is workable provided the proceed indication is available. Long transmission delays due to long distances or due to intervening store-and-forward systems would however render this approach awkward or unusable.

The error-detecting possibilities of the echo-back scheme suggest its use even when terminal lock is used. Provision to switch to half-duplex in cases of excessive echo delay is then necessary.

Interrupt

An absolutely essential feature of remote terminal operation is the "interrupt" ability. There must be a key or button whose depression causes instant detachment of the terminal from the current program stream. This interrupt must work even when the terminal is locked. The resulting status of the previously attached program is not discussed here. Normally the terminal is attached to some supervisor command module and is readied for command level input.

Reasons for needing interrupt ability include: (1) the need to stop the attached program which may for example be looping or producing meaningless printout; (2) the desire to attach the typewriter to another stream, possibly belonging to some other program.

Implementation of this interrupt feature requires either full-duplex operation of both terminal and computer, or half-duplex operation with some sort of an auxiliary, possibly narrowband independent channel. The latter is effectively provided by the teletype line-break technique of putting a "space" on the line whose duration is long enough for unique interpretation. The terminal lock must not lock the interrupt button.

Identification

All terminals must be able to identify themselves uniquely to the system. The teletype automatic answer-back scheme is a good example of this ability, because the answer-back message can be long enough not only to provide unique identification but also to independently indicate possible special terminal features.

Although user identification rather than terminal identification should normally be used to control access to the computer and to files in the File

System,[4] positive terminal identification permits default user identification and can indicate that the terminal is in fact a type known to the system.

CONNECTING AND SWITCHING REMOTE TERMINALS

As suggested in Fig. 1, remote terminals can access a computer system via a telephone central office or private branch exchange (PBX). The basic reasons why such automatic switching is advisable in a large-scale multiplex computer system involve its general flexibility and lower cost. This is especially true if continuous system availability is important. The removal from service for repair or preventative maintenance of one of a system's GIOCs, for instance, requires expensive duplication of computer ports to guarantee access to private lines.

Complete automatic switching provides:

1. User-controlled access to more than one computer. The Bell Telephone Laboratories, for instance, will have four geographically separated, large multiplex computer systems by 1967.
2. User-controlled or switching-system-controlled avoidance of unusable computer ports.
3. Static and dynamic load sharing of remote users with multiple computers.
4. Greater flexibility in planning.
5. Concentration of low-usage terminals.
6. Automatic Direct Distance Dialing access and possible use of existing tie lines between PBXs.
7. Easier terminal maintenance, because of the availability of test centers via the switched network.
8. Terminal-to-terminal communication.
9. Ability to speedily assign, connect, move, reassign, etc., terminals.

An interesting problem can arise while switching remote computer terminals through a telephone switching system. Existing telephone switching plant is engineered to handle the traffic of talkers. A crucial parameter of talker traffic statistics is the product of the average circuit holding time and the average calling rate during the "busy" hour; this quantity is typically in the range of 3-6 call-minutes. Thus an adequate number of talking paths through the switching system may be from 5-10 percent of the number of subscribers. A study made of the holding times of Project MAC users revealed an average holding time of about 1 hour; 20 percent held less than 5 minutes, 50 percent less than 30 minutes, and 80 percent less than 100 minutes.

It may therefore be difficult to add any sizable number of remote typewriters to an existing switching system without impairing telephone service, unless the terminals are to be used for only short holding time inquiries. It is possible to modify existing switching facilities or engineer new facilities at reasonable cost. This, however, can be time-consuming, and planners of remote computing systems should alert telephone companies as soon as possible.

Transmission Status

Each communication-oriented adapter channel on the GIOC can, in addition to receiving and transmitting data, sense a number of local external conditions. These sense lines will be used typically to read status information from standard telephone data sets. The system can then be fully aware of when the ringing signal is present, when data set carrier is present, when data can be sent, etc. These adapter channels also provide control outputs which can operate data set functions, such as causing a hangup.

CONCLUSION

The communication and input/output problems inherent in a large-scale multiplex computer system have been discussed. Hardware and software philosophies and a description of a general input/output controller intended to cope with these problems have been presented.

It is felt that the modular and dynamic structure of the input/output software and its flexible stream switching ability are essential to the success of a multiplex computer system. Similarly, the hardware flexibility and the uniform software approach permitted by the new controller greatly simplify the design of such computer systems.

ACKNOWLEDGMENT

The material presented here includes the thoughts and efforts of many colleagues.

REFERENCES

1. F. J. Corbató and V. A. Vyssotsky, "Introduction and Overview of the Multics System," *FJCC*, 1965.

2. E. L. Glaser, J. F. Couleur and G. A. Oliver, "System Design of a Computer for Time-Sharing Applications," this volume.

3. V. A. Vyssotsky, F. J. Corbató and R. M. Graham, "Structure of the Multics Supervisor," this volume.

4. R. C. Daley and P. G. Neumann, "A General-Purpose File System for Secondary Storage," *FJCC*, 1965.

5. E. E. David, Jr., and R. M. Fano, "Some Thoughts About the Social Implications of Accessible Computing," *FJCC*, 1965.

SOME THOUGHTS ABOUT THE SOCIAL IMPLICATIONS OF ACCESSIBLE COMPUTING*

E. E. David, Jr.
Bell Telephone Laboratories, Inc.
Murray Hill, New Jersey
and
R. M. Fano
Massachusetts Institute of Technology
Cambridge, Massachusetts

Prominent among the products of technology that have shaped our society are automobiles, electric power, and telephones. They provide us with personal transportation, with aids in our physical labor, and with convenient communication. They have radically altered the pattern of our business and private lives. Nobody will deny that these products of technology have substantially increased our mobility, have eliminated a great deal of tedious physical labor, and have contributed vital threads to the fabric of society and commerce.

Yet, they have also brought to our society ills, frustrations, and problems, few of which seem on the wane. The flight to suburbia in search of more elbow room and greenery has left a disproportionate fraction of economically and culturally underprivileged families in the cities. The same technology which has given us new dimensions in communication has been used to implement eavesdropping equipment. The same power tools and machines that are at the foundation of our industrial society caused great grief to people whose obsolete skills were their only source of livelihood and pride as working members of society. Finally, automobiles and power tools are causing us to lose our physical stamina, thereby making us easier prey for disease.

The full influence of these products of technology was felt only some years after the underlying technical advances had come to pass; namely, at about the time each of them became accessible to a large segment of the population. We are now at that stage with computers. Technical means are now available for bringing computing and information service within easy reach of every individual in a community. What will be the effect on our society?

Such service will provide to the individual "thinking tools," somewhat analogous to power tools, to aid him in his daily intellectual labor. These thinking tools will increase the power, skill, and precision of his mind, just as power tools today increase the power, precision, and skill of his muscles. As a matter of fact, there is some question whether our increasingly complex society can survive much longer without falling apart from its own weight, unless individual thinking aids become

*Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

available. At the same time, the benefits they may bring to society will unquestionably be mixed with a dose of new problems and frustrations.

The following remarks cannot help being superficial because of the great complexity of the issues involved. Not one but several papers would be required to analyze these issues to any depth. Thus, this paper is being presented primarily to stimulate discussion and further thought.

A HANDLE ON COMPLEXITY

The increasing sphere of influence of all events and human decisions is a characteristic of our society. Any change or perturbation in the status quo has reverberations which reach often into unexpected quarters. The increasing complexity of provisions embodied in our laws, regulations, and business operating procedures means that the individual has to contend more and more often with situations that he cannot personally master. Frustration and loss of time are among the least painful results.

The tax laws are a good example, as well as one of considerable importance to all of us. As a matter of fact, the tax situation of any one particular individual or business is usually rather straightforward. The difficulty lies in reducing general laws and regulations to one's own specific case. The laws and regulations must apply to a great variety of situations and their complexity is probably unavoidable. Examples, which are intended to illustrate application to common situations, are seldom useful, because they differ in some minor detail, not obviously unimportant, from the case of interest. The crux of the matter is that the number of special situations differing in some material details is so great, it would be impossible to explain for each of them the implications of the applicable laws and regulations. Even if it were possible to do so, the individual would still have the problem of finding the one applicable to his case among all possible special situations.

On the other hand, it would be perfectly feasible to write a computer program that would ask pertinent questions, in sequence, and provide necessary instructions and warnings on the basis of the answers supplied by the individual. In its simplest form, such a program would operate as a mechanized income tax form, with the important difference that it would not ask questions clearly inappropriate in view of preceding answers. Of course, computations would be made automatically on the basis of the data supplied, but this would be the least important and least helpful aspect of the program. Such a program would not have to store a dictionary of specific situations, but could work out the logical consequences of the laws and regulations in each particular instance. Where choices were available, an individual could investigate their implications in his own special case and follow the course of action most advantageous to him. One can conceive also of having the program approved by the Internal Revenue Service so that no question would exist about its correct interpretation of the law. Even further, we can envision the income tax laws and regulations being originally prepared in the form of computer programs so that legislators and Internal Revenue officials could explore more accurately and efficiently their consequences. Speculating about such matters is merely an amusing exercise, and at this time we are bound to invent merely the equivalent of a horseless carriage, rather than the modern automobile.

One can think of many other instances in our society where accessible computing service, with the appropriate software, could help individuals to contend more successfully and with less frustration with the complexities of the modern world: from paying bills and balancing one's bank account to planning a will; from budgeting the family income to selecting investments and making plans for retirement. It may seem strange at this time to envision the average man and housewife using a computer. Yet, to some people years ago it must have seemed equally inconceivable and perhaps sacrilegious to allow the average housewife to turn on powerful motors and operate such complex machines as today's automatic washing machines and driers. Not many years ago we would have winced at the thought of allowing teen-agers to spend hours monopolizing such a priceless creation of human inventiveness and technology as the telephone.

A HANDLE ON INFORMATION

Information is alarmingly plentiful these days. We are dutybound to acquire, record, search, and use it. While a great deal of effort is being spent in acquiring and recording information, our effectiveness in searching and using it still leaves much to be desired. Information has the unfortunate habit of most often being outdated, hard to locate, and re-

corded in a form poorly suited to one's needs. One reason information is often outdated is that it takes so long to collect and process it. Perhaps nothing short of a widespread information and computing service could provide an effective handle on information.

If such a service were in widespread use, information could be acquired and digested in near real-time and automatically recorded in the mass memory of the computer system. Thereby inventories, abstracts, bank balances, and on and on could then be available on a topical basis. The cost of storing information in the mass memory of a computer is still high, but not inordinately so. A page of single-spaced text stored in the disk file of the current MAC computer system costs approximately 10 cents per month. We see no reason why recording in the mass memory of a computer system should not become competitive with other recording media. With all significant actions being taken with the aid of a computer system, the contents of the system's mass memory would provide a complete, up-to-date representation of the state of the community that it serves. Technical means are not lacking for protecting private information from unauthorized access, while at the same time making it available for statistical surveys and other legitimate purposes.

Once the necessary raw data are automatically available in a computer system, we envision the development of programs to answer any well-defined queries; even those not specifically envisioned by the developers of the programs. We do not intend to imply that we or anybody else knows how to prepare such programs yet, but we do not see any major roadblock to progress in this direction. We are optimistic about technological progress, and can envision computer systems that permit communication (voice and other) interspersed with data processing. On a "conference telephone call," the third party would be a computer. Such a system would enhance, by orders of magnitude, the ability of people to interact and cooperate with one another in a manner both convenient and meaningful to each of the individuals concerned.

THE THREAT TO PRIVACY

The very power of advanced computer systems makes them a serious threat to the privacy of the individual. If every significant action is recorded in the mass memory of a community computer system, and programs are available for analyzing them, the daily activities of each individual could become open to scrutiny.

While the technical means may be available for preventing illegal searches, where will society draw the line between legal and illegal? Will the custodians of the system be able to resist pressure from government agencies, special-interest groups, and powerful individuals? And what about the custodians themselves? Can society trust them with so much power?

These are very difficult questions indeed. For many purposes, information can be depersonalized before it is put into the central file. We can devise means for providing the equivalent of safe deposit boxes for private information. A hierarchical file system, personal and modular on the lower levels, and impersonal and merged on the upper levels, is another possibility. Processing and access by other than the owner could be restricted to the upper levels. In any case, privacy can be preserved if the lower levels are left decentralized.

THE CULT OF IMPERSONALITY

The use of identification numbers and the issuing of authoritative and authoritarian instructions and answers are associated in the public mind with computers. Of course, these associations are the results of attempts, for the sake of efficiency, to fit people to the capabilities and idiosyncrasies of computers. The attempt to bring computers within easy reach of individuals is in the opposite direction. Proper names and other means of identifying individuals and locations are just as understandable to computers as identification numbers, and are much more pleasant to people. Computer programs can ask and answer questions in a very polite manner, and can even be made to chitchat realistically enough to fool a person for a little while. Computer programs don't have to be authoritarian and can be made to act unpretentiously. They can make suggestions that leave room for choice, simply warn the person that his course of action may be ill-advised, and still allow him to proceed.

There is nothing we can see inherent in the use of computers that will impersonalize, institutionalize, or automate our behavior. The danger lies in ourselves. Through mental laziness, or fear of accepting responsibility, or just plain neglect we may

delegate to computers prerogatives that should remain ours. Computers are literal-minded, as the late Norbert Wiener was never tired of pointing out. They will not take into account any premise, any limitation, or any fact that has not been made available to them. We should never delegate to them either the formulation of our problems, or decisions as to the adequacy of the solutions they produce.

Our institutions are continuously changing, and some of these changes may appear impersonal simply because they are in conflict with the customs ingrained in us from our youth. The widespread availability of a computing and information service will encourage institutions to change in new directions which may well be inconsistent with our present customs. These changes will not be required by the use of computers, but by the needs of institutions themselves. An example we can foresee concerns financial transactions.

Years ago, money consisted of gold and silver coins whose intrinsic value was identical with the nominal value marked on them. With the increasing number of financial transactions, gold coins proved to be too heavy and inconvenient and were relegated to the vaults of banks and to the strongboxes of individuals. Paper currency came into being, and with it a clear separation between the evidence of wealth and wealth itself. The value of paper currency was both guaranteed and enforced by government. Eventually, it became inadequate to the needs of private individuals and businesses, and personal checks came into use. Checks are twice removed from wealth itself, but one can still touch them and carry them in his own pocket. They are still a tangible evidence of wealth.

We are now at the threshold of a further step away from tangible wealth, in our financial transactions. With the same computer system serving banks, stores, business organizations, and private individuals, we will have available a more convenient form of implementing financial transactions. It will no longer be necessary to mail bills and return checks. Yet, each individual will always be able to have a current accounting of his financial affairs and to authorize payments by simply pressing a key. However, will people be willing to accept the reply of a computer system as evidence of their wealth? We think so, given time. But we are also mindful of the fact that many people around the world are still unwilling to accept personal or even travelers' checks, some don't trust banks and hide currency in their homes, and some refuse to accept anything but gold and silver coins.

UNEMPLOYMENT

Much has been written about unemployment that computer automation has caused, and may cause in the future. An answer often given is that computer automation will create more jobs than it will eliminate. It has been said too that a good man will always find a job, and in any case our affluent society will surely provide a more than adequate livelihood for the jobless. We think such statements miss the mark. The economic aspects of unemployment are only part of the problem. Work is not only a way of making a living, it is also the channel through which one contributes to his family and to society as a whole. Without a job one loses his self respect and the respect of those around him. This is particularly true when the job has been lost to a machine. In our present society, not only must one work to be happy, but one must also feel that he is contributing through some special skill of his own. Competing with a machine is difficult and frustrating, and so is the acquisition of new skills. The most distressing aspect of unemployment is common to the forced retirement of the man who is still physically and mentally fit. Feeling useless in an active society is a sad lot indeed.

Perhaps we can devise better ways of educating people to meet the demands of a changing world and enable them to learn new skills as older ones become obsolete. Perhaps our job-centered society must change many of its present attitudes. In any case, neither of these alternatives seems likely to provide the whole answer. Women have long been faced with "early retirement" to the household, which holds few satisfactions for many. Some women compete with men for jobs effectively; many more spend much of their creative effort in service, social and community or government. Many take up art or music or sports. When all routine and perhaps some nonroutine data collection and processing tasks are performed by computers, many men may have to make similar adjustments. Already our economy is service-oriented. The U.S. Office of Business Economics estimates that today 55 percent of U.S. jobholders are in service industries. The decline in manufacturing jobs began in 1953, but has not produced the expected unemploy-

ment because of an explosive increase in service jobs. Certainly, this is a hopeful sign, but an effort is needed to make a wider range of service jobs socially acceptable.

It has been said that many, perhaps a majority, of people in our society are incapable of anything except routine work. We are unwilling to accept this as a basic premise. Experience shows that people have vast resources, both intellectual and otherwise, which can be brought to the surface by appropriate means. We share the enthusiasm of Dr. George Gallup in the vast potential of people, as yet undeveloped.[1] The limitations we see today in the crystallized part of our population are probably more a result of their past experience than of their basic abilities. One particularly impressive piece of evidence comes from the several high school curriculum revisions undertaken since the middle 1950's. Children are now being taught in high school what their parents or older brothers and sisters were taught as sophomores in college. Typically, it has been found that children can be taught almost anything; the limitations lie in teachers who have difficulty in overcoming their past. The remarkable progress in high school education came from massive efforts in both subject matter and pedagogy. Similar efforts are underway in continuing education and retraining programs. These are vital to solving the problems of people and machines.

CONCLUSION

We do not pretend to have answers to the many questions raised here. While we have opinions which tend toward the optimistic, we take for granted that the new resources, among them computers, will be abused as well as used. We believe, however, that abuses (namely those uses which rob us of opportunity and individuality) will be recognized as such, for computers can affect our ethics, creeds, or standards only slowly compared to technological change. Preservation of these will, as always, depend upon the thoughtful and conscientious action of individuals and institutions. In the end, exploitation of computers for the benefit of society hinges upon two pivots: education, and responsible considered action by those of the technical community able to exert some influence.

REFERENCES

1. G. Gallup, *The Miracle Ahead,* Harper and Row, New York, 1964.