STRUCTURE OF THE MULTICS SUPERVISOR

V. A. Vyssotsky
Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey
and
F. J. Corbató, 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 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 complements 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 action: 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 segments (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 objects, 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 wil 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 customes 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 procsses 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 intersegment 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 ot 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 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 non-standard 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. Wolman of Bell Laboratories for allowing us to paraphrase some of his concise observations about the problem of scheduling.