MIT/LCS/TM-14

SUSPENSION OF PROCESSES

IN A MULTIPROCESSING COMPUTER SYSTEM

Carla M. Vogt

September 1970

SUSPENSION OF PROCESSES

IN A MULTIPROCESSING COMPUTER SYSTEM

Technical Memorandum 14

Carla M. Vogt

September 1970

(This report was reproduced from an M.S. Thesis, MIT,
Dept. of Electrical Engineering, February 1970.)

PROJECT MAC

Massachusetts Institute of Technology

Cambridge Massachusetts 02139

SUSPENSION OF PROCESSES

IN A MULTIPROCESSING COMPUTER SYSTEM

Carla M. Vogt

MAC Technical Memorandum 14

September 1970

(This report was reproduced from an M.S. Thesis, MIT,
Dept. of Electrical Engineering, February 1970.)

Massachusetts Institute of Technology

PROJECT MAC

545 Main Street Cambridge 02139

SUSPENSION OF PROCESSES

IN A MULTIPROCESSING COMPUTER SYSTEM

Technical Memorandum 14

Carla M. Vogt

September 1970

(This report was reproduced from an M.S. Thesis, MIT,
Dept. of Electrical Engineering, February 1970.)

PROJECT MAC

Massachusetts Institute of Technology

Cambridge Massachusetts 02139

ACKNOWLEDGMENT

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).

CONTENTS

CHAPTER ONE

SUSPENSION CAPABILITY

1.1 Introduction

In the practical operation of a computing system it is often necessary to halt the execution of a user's program so that it can be restarted later just where it left off. The simplest example is suspending execution in order to handle a hardware interrupt. Sometimes it is necessary to suspend execution for a longer period, as when a user has run out of funds or a user with higher priority desires to use the system. In interactive time-sharing systems, a user may desire to suspend his work in order to go home for the night. In all of these cases it is wasteful to destroy the work already done. Moreover, the job may have permanently altered a crucial storage file (such as the payroll records for an entire company). Hence the need for being able to suspend a job, i.e., to stop it so that it can be resumed later just where it left off.

A little thought given to the examples above will show that suspension tends to be more involved as the expected period of suspension grows longer. Saving a user's job overnight is more demanding than providing an interrupt capability, since more information about the job must be saved. Hence the term suspension in this thesis will usually refer to suspension of a user's job for a long

period, when almost all information about the job has to be saved.

Even suspension for a long period is not very difficult in a primitive computing system. The operator presses a "halt" button, jots down the machine conditions, dumps the contents of core onto tape, and saves these items together with the user's input tape and perhaps a scratch tape. To resume the job later is equally straightforward. In the complex, multiprogramming, multi-access systems being implemented today, suspension is more complex. When many users are time-sharing a single computer, suspension of one job must not affect others adversely. In addition, as user jobs become less self-contained, it becomes more difficult to ensure their resumption.

We say that a system has a <u>suspension capability</u> if it is able to suspend a user's job so that other jobs are not harmed and the suspended job can be safely resumed later. This thesis is concerned with identifying some of the implications of a suspension requirement for large, general-purpose time-sharing systems. We begin by getting our subject into focus. What is the entity that is suspended? Why should suspension be a problem in advanced systems? What are the requirements for a suspension capability in such a system?

1.2 Computation and Process

"Job" is a vague word for the execution of a user's program in the context of a computing system. We replace the word "job" with a concept which more accurately defines just what is being suspended. The following definitions are based on the discussions in Dennis and Van Horn[1] and Van Horn[2].

A computation is the execution of a set of actions which are partially ordered in time and which manipulate certain data variables, some of which may be input to or output from the computation. The ordered set of actions is called a program. A process is a totally ordered subset of a computation, i.e., the execution of a sequence of actions within the computation. If a program describes a sequence of actions it is called a sequential program. If it is not restricted to a single sequence of actions it is called a multiprocess program and the execution of the program is called a multiprocess computation. If we assume that the partial ordering of actions in a multiprocess program is the only ordering enforced on the computation, then the computation is said to be asynchronous. The various computations going on at any one time in a multi-access system are said to run asynchronously of each other, since they are separately programmed.

A computing system is a collection of hardware and software resources (this concept is further developed in

the next chapter.) A computing facility is a computing system in operation, that is, with one or more computations running on it.

In this thesis we make the simplifying assumption that any computation to be suspended consists of a single process. There are two justifications for this. First, almost all present-day general purpose computing systems support only single process computations, although systems are being designed to support multiprocess programs. Second, the first step in learning how to suspend and resume a multiprocess computation is learning how to suspend and resume its constituent processes.

1.3 Communication between processes

A system has a suspension capability if it can halt the execution of a user's process in such a way that other users are not adversely affected, and that the process can successfully resume execution later. Let us consider why suspension should be a problem at all.

If a collection of processes is executing concurrently on a computer system, and one of the processes stops, how could other processes be adversely affected? There are two ways in which processes affect--i.e., communicate with--each other. One we may call control, that is, causing processes to begin or halt execution. The other is through sharing of common resources. If process P is suspended, it may fail to

make expected decisions that other processes should begin execution. After P has been suspended another process may decide that P should execute, not realizing that P has been suspended. The thesis will be only minimally concerned with control problems.

Beside these control problems are the problems which arise because resources are shared. Here we roughly define resource as anything a process needs to run (a more precise definition can be found in Chapter Two). Suppose the process has a tape mounted on a tape drive when it is suspended. In a system which manages tape drive allocation within the supervisor, no other process will be able to use the tape drive.

What can prevent a process from resuming execution? Two recent developments in systems design make successful resumption of a process more difficult.

First, one of the recent fundamental advances in computing systems is the advent of direct sharing of information. In earlier systems there was either no sharing of information or copies were shared. Corbato´ and Saltzer[3] have shown that direct sharing is desirable to eliminate the need for copies and the difficulty of updating that multiple copies imply. The Multics system[4,5,6,7,8,9], implemented at Project MAC, is one of the first systems to incorporate direct sharing of programs and data by several users. While direct sharing represents an advance in

computing systems design, it also poses problems. In particular, the number of interfaces between different user processes increases enormously. Increased communication between programs increases interdependency. As a result it is no longer so simple to suspend a process and resume it several days or weeks later. For example, suppose one user "borrows" a program from another user. In a system with direct sharing both borrower and owner use the same copy. If the borrower's process is suspended while executing the borrowed program, and if the owner meanwhile decides to change the program, the borrower's process cannot be resumed.

A second development in computing systems is the advent of large, helpful systems which provide a multitude of services. In such systems users tend to become heavily dependent on system supervisor and utility programs. The user may be considered to be "borrowing" such system programs. The consequences of system changes are as drastic to suspended user processes as the consequences of a change in the processor's instruction set would be to user programs in a more primitive system.

1.4 Suspension Capability

We are now able to give a more precise definition of suspension capability. We said that a system has a suspension capability if it can halt the execution of a user's

process in such a way that

(1) other processes are not adversely affected (even if the suspended process never resumes execution),

(2*) the process can successfully resume execution later.

What does it mean to "successfully" resume execution? It means to resume execution in such a way that the resumed process is not adversely affected by any changes which have occurred since the process was suspended. The changes in question may have been control signals or changes to resources needed by the process. If we ignore problems of control we might require the system to guarantee that the resources needed by the process will be available when the user wants to resume the process.

But the system cannot always guarantee that a suspended process can be resumed. First, the user himself may delete a program or data table used by his process. Therefore we partition the resources available to processes into those whose modification or deletion is controlled by system policy and those in control of users. Then a more reasonable requirement on the system is the ability to guarantee that resources controlled by the system are available when the user wants to resume the process.

Secondly, resumption of a process tends to become more difficult as time increases, because of the increased

probability of changes to resources. As explained by Corbató and Saltzer[3], as well as others, future computing systems must be able to evolve and adapt to changing conditions. In such systems the system itself undergoes frequent modification. It is still possible, however, for system modification to be guided by a policy which ensures a high probability of resumption within some known T time units after suspension. Requirement (2*) above may therefore be replaced by the following:

(2) some suspended processes can be resumed; namely, those unaffected by any changes which may have occurred during the period of suspension.

(3) the system can decide whether resumption is possible, so that processes are not incorrectly resumed, with erroneous results.

(4) system policy is formulated to guarantee a high probability (near unity) that a process may be resumed at any time within a given T time units after suspension, for some "acceptable" value of T, despite changes to system resources, as long as no user resources are changed.

The implications of suspension on control and allocation of resources are interesting in their own right. But this thesis will be concerned primarily with requirements (3) and (4).

1.5 Organization of the thesis

The foregoing discussion has served to bring the subject of the thesis into focus. The thesis is concerned with exploring some requirements for a suspension capability and with system design and system policy for fulfilling those requirements.

Since we are concerned with system resources we develop a model of a computing system (Chapter Two) as a collection of resources. In Chapter Three we explore the nature of a process's needs for resources. Chapter Four describes some implications for system policy on resource modification. Chapter Five is a discussion of some problems of suspension in the Multics system. Chapter Six presents some conclusions of the investigation.

Throughout the thesis examples are drawn from the Multics system, for the reason that Multics makes explicit the difficulties raised by suspension. The examples assume some familiarity with the Multics system, as described in the references[4,5,6,7,8,9]. Also useful are the introductory chapters of Saltzer[10] and the discussion by Bensoussan et al. of the Multics virtual memory[11]. Anyone who is neither a speed reader nor already a Multics initiate can skip Chapter Five. The remainder of the thesis should be understandable even without the examples.

CHAPTER TWO

THE SYSTEM

2.1 Introduction

Since the suspension problem has to do with the modification of resources in a computing system, we need some insight into computing systems in terms of the needs for and allocation of resources in processes. The purpose of this chapter is to present a model of a modern computing system so that requirements for suspension can be discussed in terms of the model.

The model is not intended to be a sketch of the Multics system. Rather, it is an independently conceived abstraction, which we will apply, as a Procrustean bed, to the Multics system. However, we do make some general assumptions about systems to which this model can apply. First, the system is assumed to have a modular design, i.e., to be made up of distinct units of program (and data), each unit having responsibility for some aspect of system functioning. These modules are more closely akin to closed subroutines than to the blocks of an Algol program: usually a module has fulfilled its function just when control in the process leaves the "return" instruction, and the module can be "called" from any other program module. Second, we assume that the system is large and complex. Third, the system supports more than one process

concurrently. These processes can and do share resources directly, and the system must regulate the shared use of resources. Fourth, the system is a multi-access system in which resources may "belong" to different users, who control the sharing and modification of resources among themselves. Last, but not least, we assume that suspension of a process is a possibility which can occur in normal system operation.

Some familiarity with the Multics system is assumed on the reader's part, and examples in this chapter will be drawn from that system. However, any other system satisfying the above assumptions would provide equally good examples.

2.2 Resources

To begin with, a computing system may be regarded as a collection of resources which a process can use. These resources can be classified as physical or abstract, and as program or data variables. Examples are shown in the figure.

	program	data
Physical	processor instruction logic	tapes, processor registers, core word
Abstract	software program	page table, process list

Every resource in a computing system falls into one of these categories. Data is just information which is not intended

to describe a computation, i.e., nonexecutable. A resource such as a table of Bessel functions or processor logic is rarely modified, but can still be considered as a variable. A page table or a program being debugged is modified more often.

The word "variable" in the above discussion might be replaced by "module" or "segment". It is a unit defined by the interfaces it presents to the outside world, viz., its functional specification. If its value changes it is the same variable. If its functional specification changes it is a new, distinct variable.

The concept of program is crucial to this discussion, and therefore deserves close investigation. We are concerned here with program modules, that is, groups of one or more external procedures (in the PL/I sense) that cooperate to perform a common function. That a module in this sense is partitioned into smaller units is of no concern here. Hence we speak of control entering a module (or a program) and of the program module returning. A program module will sometimes be called a program resource, in order to emphasize its usefulness to the process executing it.

It might be possible to establish criteria for what constitutes a separate program module, from the point of view of the system as a collection of resources. Instead we assume that the program modules are given, and accept them as given.

2.3 A lattice of resources

In a large and complex system a collection of resources, even when catalogued, is not much of a model. What is needed is structure. Edsgar Dijkstra[12,13] and, drawing from his insights, Randall and Zurcher[15] and Parnas[14], have proposed that systems be hierarchically designed, so that programs in the system can be arranged in a lattice* in the following way. The programs at the bottom of the lattice are completely self-contained. We say that they form the zeroeth layer of the system. In the first layer are programs that rely only on the programs of the zeroeth layer. Programs in the second layer rely on the zeroeth and first layers, and so on. The arcs of the lattice show the dependence relation.

When a system is so constructed, the ith layer acts as an interpreter for the (i+1)st layer. The (i+1)st layer is, as it were, programmed to execute on a "machine" provided by layers 0 through i.

*A lattice is a set of objects on which a partial ordering relation, often denoted $\leq$, is defined. A partial ordering relation is reflexive, transitive, and anti-symmetric. If a and b are two objects in the lattice, it is not necessary that either $a \leq b$ or $b \leq a$. The partial ordering in this case is the dependence relation that occurs when one program calls another or is coded with the understanding that another program may "help" during its execution (e.g., paging).

Even when a system has not been hierarchically designed it is possible to view it hierarchically. The Multics system was designed with a trace of the notion (viz., the notions of pseudo-processor[10] and virtual memory[11]). But for the most part, the system is thought of as two-level: software program and hardware that executes the program. We will view the Multics system somewhat differently. To begin with, a processor instruction is itself a program. If the processor is modularly designed, these programs may be regarded as occupying distinct levels of system. (This is most clearly exemplified in microprogrammed processors.) Rather than thinking of control as being "in" just one program, plus just one instruction, we view a whole group of programs (both hardware and software) as being "current" at any given time, no more than one per level of system. (This assumption is unwarranted in systems which allow user handling of faults--see section 5.4 on user faults.)

It is possible to view Multics in this way, even though it was not so designed, because of its use of external procedures and especially because of its modular design. Further, the attention given in the first design and in subsequent redesigns to simplicity in the system has tended to result in a more hierarchical pattern of dependence and fewer complex interrelationships between programs.

"Dependence" should be more precisely defined. One program depends on another if is is coded to call that program in some circumstances, or if it is coded with the understanding that control may trap to that program in some circumstances. Dependency in this sense provides just the partial ordering relationship we need in constructing the system lattice.

Three anomalies present themselves here. First, some programs transfer control to other programs but can't be viewed as being interpreted by these programs, as required by the model. Second, sometimes two modules call each other. Third, what happened to data variables in this "lattice of resources"?

There are certain programs in some systems that don't fit easily into the picture given above. An example is the Multics Shell. This program acts as a dispatcher, interpreting command lines typed by a user and causing the requested command program to be executed. The Shell itself does not include programmed calls to any commands, yet it causes them to be called. In this case we do not consider that the Shell depends on the command, becasee completion of the Shell's work is not dependent on the existence or proper functioning of the command program. The Shell is coded so that its job may be considered finished when it has called the command. A similar case is provided by the fault interceptor in Multics, which gains control whenever

a fault occurs, and causes the appropriate fault handlers to execute. As the fault interceptor acts as a dispatcher, it can successfully perform its function even if one of the fault handlers is in error. One of the fault handlers in Multics is the divide check handler. A user may substitute his own divide check handler for the one provided by the system. The fault interceptor does not depend on the handler. In summary, a program whose function is dispatching (as opposed to, say, calculation of a trigonometric function) is not ipso facto dependent on programs to which it happens to dispatch control.

The second anomaly is the phenomenon of two program modules which call on each other. We do not ask whether the modules are necessarily mutually dependent. That is, if execution path A in program X is dependent on program Y, and Y is dependent on path B in X, then do paths A and B in fact intersect? This is a question to be answered in the design of the system when the supervisor is divided into programs. Given the programs as they happen to be, we revise our notion of what constitutes a node of the lattice. A node is a maximal set of mutually dependent program modules.

The third anomaly is that the system--a collection of resources--has been layered without any mention of data variable resources. Once program resources have been ordered in a lattice structure the data variable resources

are easily included. Namely, a data variable is needed by a program node if the program uses the data variable. If the program may modify the variable it is write-dependent on the variable; otherwise it is read-dependent on the variable.

We wish to give a precise definition of the dependence of a program on a non-program resource. It would seem that a program which makes the call

write ("cardpunch", data_area, 1, 100)

to punch a 100-card file is dependent in some sense on the existence and availability of a card punch. However, it is possible that "the system may decide" merely to queue a request for use of the cardpunch if none is available. Indeed, the program which contains that call is actually dependent on the "write" program and not on the card punch. At some lower level of the system, however, a program may exist which issues a connect to an I/O controller to cause punching to begin. This program is dependent on the existence and availability of a card punch.

This last example suggests another way in which a program is dependent on a data variable. The page fault handler may run to completion, but if it obtains incorrect information about the whereabouts of the page on secondary storage, it will perform its task incorrectly. We do not in this case consider the page fault handler to be dependent on the information per se, i.e., on some particular

value of the variable, but rather on the data variable, and we make the simplifying assumption that all variables have the correct value.

Any process not solely dedicated to system functions or serving as a desk calculator is probably making use of resources not considered so far but important to suspension, that is, user supplied resources. A computation running on behalf of a certain user may need programs and data variables which are modifiable by that user or other users. These resources are made accessible to the process through the computing system, and may be considered as system extensions, or resources in the extended system. In fact, the chief differences (from our point of view) between these resources and system resources are: first, that system design cannot assume any maximum time limit on the execution of a user program (since it may contain a tight loop); and second, that modifications to user programs and data cannot be controlled by system policy. Because the system cannot rely on user programs or data, it follows that no user program or data variable may be on a level below any system program or data variable. That does not prevent the inclusion of programs in the system which dispatch control to user programs, but it does require recognition that those programs may never return.

2.4 State of a process

We now connect this model of a system as a collection of resources to the earlier description of a process as the execution of a program or sequence of programs. At any point in real time the process is "occurring" at various levels of the system (I will say "system" rather than the bulkier "extended system" where the meaning is clear.) That is, the process is described by one or more programs which have been partly executed but not completed. These programs may represent several layers of system and many non-program resources. As an example, consider a Multics process which is executing at each of the following levels: (a) the processor is performing a fetch, (b) as part of an lda (load accumulator) instruction, (c) which occurs in the page fault handler, (d) processing a page fault for the list command program. At each level the process is using program and other resources. At some level the process may be altering the state of certain resources. In this example, the instruction is modifying the accumulator register and the page fault handler is modifying a page table. We say the process is write-dependent on resources it may be modifying and read-dependent on others.(see discussion of allocation, below). If we know which programs are current, that is, which program resources the process is using, and the resources needed by each program, then we can begin to characterize the resources needed by the computation at any

instant of real time. This will prove to be useful in designing a suspension capability, and we will return to it later.

In summary, the needs of a process can (to a first approximation) be characterized by the description of the current program modules and the data variables needed by each module.

2.5 States of resources and state transitions

Sometimes what might at first appear to be two distinct data variables in a computer system turn out, on closer inspection, to be better regarded as parts of a compound variable. As an example from Multics, consider an entry in the core map and the contents of the associated 1024-word block of core. If the core block were replaced with a page from secondary storage, and the core map were not updated, the system must be considered to be in an inconsistent state. A typical compound variable occurs when one data variable is used to describe the state of another variable. This compound variable, like a simple variable (e.g., bit or word), has a (possibly large) number of well-defined states. When a program node of the system resource lattice is write-dependent on a certain compound variable, we may say that the program module performs state transitions on the variable. While it is undergoing state transitions the variable is in an inconsistent state.

We tend to find more complexly structured compound variables as we ascend the resource lattice. On the level of the machine instruction the variable is typically a word or a register. At higher levels it may be a variable-length list with a count, or even (to use an example from Multics) a segment and its directory entry. Although any degree of complexity is possible immediately above the machine instruction level, in practice the variables tend to become more complex as a more sophisticated "machine" is available to perform state transitions.

Because compound variables may be very complex they may not be easy to identify. A frequent clue to the presence of a compound variable is a lockword that regulates access to the variable, or a felt need for such a lockword.

2.6 Sharing resources

A modern computing system supports multi-processing in which the processes share access to variables. This causes a problem when the system contains compound variables. Suppose a certain compound variable, composed of parts A and B, is undergoing a state transition involving both A and B. First A is modified, then while the resource is in an inconsistent state another process tries to use the inconsistent data, resulting in an error.

In order to avoid this type of error, systems introduce some regulation into the sharing of resources.

An obvious method (once it had been pointed out by Dennis[1]) is as follows. To modify a variable a process must have exclusive use of, or write capability for, the variable. To read a variable a process has read capability for the variable, thus preventing any other process from getting write capability for the variable. Thus a process has just one kind of capability for a variable: none, read, or write. Similarly, we say that a variable is attached to a process which has read capability for the variable, and assigned to a process which has write capability for it. A resource may be assigned to one process, or attached to one or more processes, but not both. We say that a resource is allocated to a process if it is either attached or assigned to the process.

We note in passing that attachment and assignment must of necessity be for a limited time only for any sharable resource. When a process has completed an operation with an assigned variable, it unassigns the variable, and the variable remains attached to the process. When it detaches the variable, it no longer has any capability for the variable.

2.7 Positioning resources

It is important to distinguish between capability and availability. Although the word "capability" seems to imply that a process with write capability can actually

modify a variable, in fact it only may (has permission to) modify the variable, i.e., as soon as the variable is made available (e.g., by paging).

As we distinguish between capability and availability, so we distinguish between giving a capability to a process (allocation) and giving a process the ability to reach a variable (positioning the variable). Sometimes the actions of allocation and positioning are identical. In Multics, for example, it is only possible for one process at a time to use the processor. Other processes do not refrain from using it before it is allocated to them. Allocation and positioning both take place when one process executes a load-descriptor-base-register instruction in favor of another process.

Often allocation and positioning are not identical. For example, Dennis[16] proposes a system in which locking conventions would be coded into the processor hardware, just as locking conventions are now coded into Multics segment control. In this case, processor hardware refrains from modifying a word in memory if that word is allocated to a different process.

Even when a process has deallocated a resource such as a tape or tape drive, the resource must still be returned or unpositioned to its original state. In Multics, for example, a process is responsible for returning tapes when it is finished with them. However, sometimes other

processes take over the responsibility of returning or unpositioning resources. An example is provided by the Multics paging algorithm; other processes clear away from core memory the no longer used pages of a suspended process. In either case, when a process is suspended, its resources must be both deallocated and unpositioned, so that its suspension does not adversely affect other processes which might need those resources.

2.8 Summary

This chapter has discussed several features of systems in terms of a rather simple model of a computing system. At this point it may be advantageous to pause and briefly review the model.

A computing system is a collection of resources, i.e., program and data variables. These variables may be organized into a lattice structure according to the dependency relationship. A program is dependent on another program if its execution implies or may imply the execution of that program. Two mutually dependent programs are considered to occupy a single node of the lattice. A program node is dependent on a data node if the program either reads or writes the data variable.

The resulting model is useful for examining and describing the use of resources by a single process, and therefore the sharing of resources by several processes.

On the other hand, it is a static model and provides no more than a basis for describing dynamically an actual computing facility on which, say, three processes are executing concurrently.

CHAPTER THREE

PROCESS NEEDS

3.1 Introduction

With the insight gained in the previous chapter we can restate the thesis problem. A program in execution has certain resource needs; the program itself, certain data variables, interfaces with other programs. If the execution is suspended, during the interval of suspension some needed resources may be modified in such a way that resumption of execution is impossible, or produces incorrect or meaningless results.

The purpose of this chapter is to see how a process needs its various resources, which changes affect resumption and which do not. In investigating the resource needs of a process we thus come to an understanding of what constitutes "safe" resumption and what does not. Such an understanding is the sine qua non for deciding whether a given suspended process can be resumed.

In the subsequent discussion we make two strong assumptions. First, we assume that all programs are correct, that is, they satisfy their functional specifications and leave compound variables in consistent states, which likewise satisfy functional specifications. Second, we assume that when a process is executing it has attached or assigned to itself all the resources which it needs,

and that this allocation of resources follows the rules outlined in Chapter Two. The main reason for this assumption is that it serves to clarify the issues by clearly demarcating when a process is liable to lose needed resources: during execution the resources cannot be snatched away; when the process is suspended it releases the resources and accepts the risk of being unable to resume execution. If we do not assume allocation of resources, it is very difficult to distinguish between problems caused by suspension and problems caused by uncontrolled sharing.

A second reason for the assumption that sharing is regulated as described is that Van Horn[2] has shown the necessity of implementing such regulation in the system hardware. It is reasonable to expect that regulation of sharing will become standard practice in future systems. Multics as currently implemented does not satisfy this assumption. The hardcore supervisor uses software locking conventions, and a locker routine is provided for user processes which carry out preprogrammed sharing. However, most user sharing is not so programmed. For example, one user may borrow a program from another user, i.e., arrange to use the program. While the program is being executed in his process, the "owner" of the program may absentmindedly delete it, causing drastic consequences to his friend's process. It is possible that later implementations will remedy this defect, perhaps following the example

of a machine proposed by Dennis[16].

3.2 Types of need

A process needs a resource to exist (in a certain state) and to be available. For example, execution of most instructions in Multics requires the descriptor segment to exist (and truly reflect the location of segments) and to originate at the word pointed to by the descriptor base-register. The problems of positioning resources are considerable, and general solutions will no doubt place important constraints on operating system design. This thesis, however, will not attempt to deal with the problems of positioning, but rather with the state in which a needed resource must exist. By "state", then, we mean not disposition, location, etc,, but content and interface.

The interfaces of a variable with other variables define it as a module. The referenceable items of a table, the functional definition and calling sequence(s) of a program, the calls and references to data made by a program, consistency constraints on the allowed values of compound variables--these constitute its interface with the outside world, and hence its place in the system and its definition as a module. Its content consists, in the case of a program, in the algorithm described by the program, and in the case of data, in the information contained in the

referenceable items.

We recall from Chapter Two that when a variable changes its interfaces it ceases to be the same variable. Two variables can be more or less closely related: for example, two procedures whose only difference is that one makes an additional call are closely related. In contrast, a list-structured data base (assuming that its structure is an interface, i.e., referencing programs "know" its structure) is utterly unrelated to a matrix-structured data base, even if both contain the same infommation. We say that a variable is modified if only its content changes. If its interfaces change, we say it has been replaced by a distinct variable.

When a process needs a variable, it may need the interface of the variable or the content or both. These three modes of need will be demonstrated in the ensuing discussion.

3.3 Allocated variables

When a process is suspended, several variables may currently be allocated to the process. These variables must be detached, of course, and reallocated to the process on resumption. The list of these resources constitutes a first approximation to the description of the process's needs.

There are three types of allocated variables. First,

the programs which are current at suspension. It might be possible to define a class of transformations to the interfaces and content of a program such that, given the locus of control in the program at suspension, transformations in that class would not affect resumption. For example, if control is about to enter the "return" instruction of the program, any transformation will do, as long as control in the modified program is also about to enter the return instruction. Specification of such a class of transformation is of dubious practicality and in any case beyond the scope of this thesis. We therefore assume that if a program is allocated (current) at suspension time, the process needs the same program (same interfaces, same contents) to resume execution.

A second class of allocated variables are data variables needed by current programs. The data variables allocated to the process represent only a fraction of those which the programs might need, because actual need at any instant depends on the locus of control in the program. If a variable is attached to the process then it requires that the variable not be modified. That is the meaning of read dependency (cf. Chapter Two). Therefore both content and interface are needed.

The third class of allocated variable is the static private (to the process) variable. This contains some information of lasting interest to one or more programs of

the process, hence it remains allocated to the process although no program which uses it is current. An example from Multics is the Known Segment Table. Another is the internal or external static storage (in the PL/I sense) for any program(s). For these variables the content is necessary. These variables must also satisfy some interface constraints as indicated in the next section.

3.4 Interfaces

The system lattice as described in Chapter Two portrays the interfaces of system program and data variables. Suppose that a certain program is current when a process is suspended, and that one of its needed interfaces changes. For example, the calling sequence or functional specification of some program it calls might change. The suspended process then cannot run properly with the modified interface.

That consideration shows that the needs of a process include not only the programs and data variables attached to the process, but also the following:

(1) all data variables on which current programs are dependent (but not attached)

(2) all program variables on which current programs are dependent (but not attached)

(3) all program variables which are dependent on static private data attached to the process.

These three constitute not a necessary but a sufficient set of variables for resumption of the process. The reason for using a larger set than might be necessary is the general impossibility of predicting the future course, and hence the future needs, of a program in execution.

In what way does the process need the three sets of variables? Clearly it is the interfaces rather than the current content of any which are needed, since by the assumption that all programs are correct the content satisfies functional specification (which is an interface) and if the content itself were needed the variable would be allocated to the process. So if one of these variables is modified but not replaced (i.e., none of its interfaces changes) the process can use the modified version. Even if some interfaces change, but no interface needed by the process changes, the process can use the modified version. Thus some "relatives" of needed modules can be used. In case (2), any "related" program may be used which has the same needed calling sequence(s) and functional specification. In case (3) any program which has the same interface with and makes the same use of the static data can be used.

One other observation may be made about private static data. What is needed is the content rather than the interfaces of the data variable. An example is the Multics Known Segment Table (cf. discussion in Chapter Five)

which contains the association between segment numbers and pathnames of segments known to the process. The interfaces to this information are of no interest if no program that uses the Table is current. So the same information in any other form would be sufficient, as long as the resumed process uses programs with the correct interface to the new data variable.

3.5 Conclusion

The preceding discussion has attempted to show the resource needs of a process. A note of caution is in order. The discussion assumes that all "variables", "programs" and "data", are nodes of the system lattice. The same discussion might not apply to programs as individually compiled, because mutual dependence of programs introduces ties which, under our assumptions, do not have to be considered. One advantage, in fact, of avoiding large, complex nodes is that the replacement of modules is thereby made simpler to comprehend.

Once it is possible to formulate, that is, to compute the needs of a given process, it is possible to compare those needs with the state of the system to find out whether the needed resources exist, i.e., whether the process can be resumed.

It may be that a given process cannot be resumed because some needed user resource has been modified or

deleted. The system cannot prevent this. It can, however, prevent system changes from interfering with resumption. Whenever possible, a resumed process should use the new version of modified modules. The next chapter discusses what may be done when a new module or version of a module cannot be used.

CHAPTER FOUR

SYSTEM MODIFICATION

4.1 Introduction

One requirement for a suspension capability is the formulation of a policy for system modification. System design is responsible for defining the locus of policy decisions and provides the means of implementing them. Often a system is designed to enable a particular kind of policy to be enforced, with system administrators given the responsibility of assigning an appropriate value to certain variables in the design. This chapter outlines a strategy for system modification to allow suspension. The variable in the strategy is T, the length of time after suspension of a process during which the system supports resumption of the process.

We desire to formulate a strategy for system modification which guarantees resumability within time T. One way to guarantee resumability is to refrain from making any change to the system which would prevent the resumption of any suspended process. That is, when a change is proposed, it is periodically compared to the needs of all suspended processes, and implemented only when no suspended process is endangered. This strategy is unacceptable, because it can result in indefinitely long delays in improvements and corrections to the system, as well as huge

administrative headaches. What is needed is a strategy in which system changes are made independently of the needs of currently suspended processes, that is, a strategy which works in every situation.

That constraint implies that the procedure to be followed when a module is modified or replaced may depend on the nature of the module but definitely not on the current state of any processes. The procedure to be followed in resuming a process depends on whether needed modules have been modified or replaced. The strategy for replacing modules therefore consists of two parts, that followed in system modification and that followed in process resumption. Cooperation between the two activities is possible if each module in the system is uniquely identified; its interfaces are uniquely identified; a "time-last-modified" tag is associated with each variable; and system modification is understood to include correct updating of these items.

We first consider the procedure to be followed when a module is modified or replaced. Since we do not wish to consider the needs of all suspended processes, we must assume that some process needs any module which is changed and that it cannot use the new version. (Later in the discussion we will see that some exceptions to the general rule may be made for special modules.)

To resume a process, we observe that if it were

always possible to use the latest version of every system module no formulation of strategy would be needed at all. To minimize complexity, therefore, we will use latest versions wherever possible, and then formulate strategy for the remaining cases. The discussion of a process's needs presented in Chapter Three implies that the suspended process may need the content andor one or more interfaces to system modules and that the modules may have been modified (content changed) or replaced (interface(s) changed). By comparing the process's needs with the information associated with each module, we can ascertain whether the process can use the latest version of the module.

The remainder of the chapter concerns what to do if the latest version cannot be used. There are two techniques: use old versions, or make sure no suspended process ever needs a given module. We discuss these techniques, showing how the basic strategy outlined above can incorporate these techniques to ensure resumption of processes that cannot use the latest versions of needed modules.

4.2 Using old versions

One possibility when a new version cannot be used is to use the old version. This is possible if the old module interfaces correctly with other modules and with other processes. Chapter Two presents a system model that exposes the interface between modules. That model does not

indicate any interfaces between processes. However, in any multiprocessing system it is possible to program the various processes to cooperate with one another, as, for example, in observing locking conventions. Sometimes the processes are programmed to cooperate in carrying out an algorithm. An example is Multics page control, which works as programmed only if all processes use the same paging program. This is an example of an interprocess interface which prohibits using an old version of a program. (For further discussion of this example, see Chapter Five.)

The requirement of correct interfaces with other variables in the same process is more manageable. We can call an old version of a module safe if

(1) it is not dependent on any other variable; or

(2) no variable on which it depends has changed; or

(3) there exists a safe version of any variable on which it depends.

If the system preserves safe versions of changed programs then it is in principle possible for a suspended process to be resumed using the safe versions.

This strategy imposes some serious constraints on system design as well as on system management. First, the process must be able to use the safe version of a program as well as the modified version. The safe version is used because it is current or needed by a current program. The modified version may be needed by a new system

program that will be invoked later in the process. So both must be able to coexist in the same process!

Second, significant bookkeeping is required in order to know when a variable is safe: to know whether a high level variable is safe it may be necessary to find safe versions of variables on many lower levels.

Third, safe versions of variables must be retained until no longer needed. If the system guarantees a near unity probability of resumption within T time units, then ordinarily safe versions need be retained for only T time units. It is conceivable, however, that a process might be suspended and resumed several times, so that its total lifetime is several times T. The system administration must then decide whether to advertise that such processes are not supported or, if such processes are needed, to extend the time for which safe versions of variables are retained.

We now pause to review what has been done so far. Whenever a system module is changed, system modification policy must take into account that some process might not be able to use the new version or replacement of the module. If a system is designed according to certain strong constraints, then for most modules it is possible to preserve the old version of the module as a safe version. For a certain class of programs this is not possible, i.e., those whose interprocess interfaces require that the same

version be used in every process. We now consider how to deal with such modules.

4.3 Delaying suspension

When neither the old version nor the new version (or replacement) of a module can be used, it is awkward for a process to be suspended while the module is needed.

The obvious solution to the difficulty is to postpone suspension until the module is no longer needed. We may distinguish two cases: the module may be allocated, or some interface to the module may be needed. In the first case, it is not difficult for the system to "know" that the module is needed and to refrain from suspending the process. For example, allocating such a module could automatically add one to a counter associated with the process. Deallocating the module could decrease the counter. If an attempt were made to suspend the process while the counter was non-zero, suspension would not take effect until the counter's value was zero.

Suspension should not, however, be subject to arbitrary delays. We wish to make constraints on any shared module which inhibits suspension while it is allocated to the process. This is equivalent to putting a limit on the execution time of very sensitive programs (since sensitive shared data variables are allocated only by such programs and private static data is by definition not shared).

We define for each program node an execution time, representing the maximum length of real time that program may be current. If the execution time of a program is indeterminate, the program is said to be unreliable; otherwise the program is said to be reliably finite. If the execution time is less than E, the program is said to be E-reliable, or just reliable. (Of course, we assume that all programs which we wish to classify as reliably finite are bug-free!)

If Q is the maximum tolerable time for which suspension can be postponed, then we may constrain all very sensitive (in the sense that suspension is awkward if the module is needed) programs to be Q-reliable. Then when a process has been chosen for suspension, the system can allow it to execute for a time (less than Q) until none of those sensitive modules are current.

We stated above that suspension should not be subject to arbitrary delays. More precisely, there are situations in which suspension must take place very quickly, for example, when a user has run out of funds or when the system is being shut down in an emergency. For these situations, it is necessary that a value for Q be determined and enforced on the very sensitive modules.

In other cases, however, suspension can be delayed for a longer time. Indeed, there is often an advantage to be gained by allowing a process to run on before

suspension. There is a cost associated with preserving old versions of modules, a cost of bookkeeping and, more especially, of storage. It may be possible to reduce that cost by delaying suspension until the process is dependent on very few modules.

First we observe that if we know which old versions of modules are needed by suspended processes we can delete the rest. But the old versions needed by suspended processes are just those which were allocated to the processes and the safe modules on which those depend. If, when a process's resources are deallocated on suspension, they are marked as needed by that process, then unneeded old versions can be deleted. Then it is possible to associate the cost of storage with the process.

Postponing suspension can now be seen to be advantageous if the cost of the delay (the processor costs, disadvantage to other users, etc., may be reflected in a price associated with postponing suspension) is less than the cost of keeping old versions of modules.

Lower programs in the system lattice have smaller execution times than programs which depend on (call) them. Therefore, it is possible to define reliability zones on the lattice, including those programs whose execution times fall within a certain bracket of times. One interesting reliability zone consists of programs with execution time between zero and Q (maximum tolerable time for emergency

suspension). The programs in this zone need never be needed by a suspended process, and old versions of these programs can always be discarded.

From the point of view of the system, user programs cannot be considered reliable. However, a user may believe that his program is N-reliable and request that suspension of his process be delayed until either the process "runs out of" the reliability zone bracketing execution times, say, Q to N, or suspension has been delayed N time units, whichever occurs first. Using such a facility the user could avoid the costs associated with saving the program for his process.

4.4 A difficult case

We have said that postponement of suspension can ensure that a process is not suspended while any module is allocated to it for which an old version cannot be used (viz., because every process must use the same version). However, such a module may be needed by a process even when not allocated to the process, i.e., when an interface to the module is needed.

For example, suppose a process is suspended while program P is current. Program P depends on R, a program for which the same version must appear in every process. While the process is suspended, R is replaced by S, which uses a different declaration for one argument. The process

can no longer be safely resumed, since it cannot use R if other processes use S, yet it cannot use S.

Changing interfaces poses difficult problems under any circumstances. Programs have to be reprogrammed to use a new calling sequence or to reflect a modified functional specification. But the special difficulty for a suspended process is that a current program may have to be reprogrammed! We describe three possible approaches to handling this case.

One approach would be to require that all programs which call very sensitive (in the above sense) variables must be reliable. This is a very serious constraint on the system. Its implications have not been explored.

A second approach would be to alter the state of the suspended process so that it has the proper interface with the new version of the needed module. This strategy requires a better understanding of programs and processes than now exists.

A third approach would be to examine interprocess interfaces of modules more closely to see under what circumstances two different modules can be used.

4.5 Summary

The system can, if properly designed, permit a process to resume execution with old, "safe" versions of needed modules. For some system modules this may be

impossible, hence it is desirable to make some programs reliable and postpone suspension until they are no longer current. The device of postponing suspension can also be used to eliminate the need of the process for some other system or user modules, and thus to reduce the number of old versions which must be retained in storage.

CHAPTER FIVE

SUSPENSION IN MULTICS

5.1 Introduction

This chapter presents some examples of suspension in Multics. The first few are suspension-like phenomena, not at all unique to Multics, but included as illustrations of the suspension phenomenon. The chapter ends with a discussion of suspension in the usual sense in Multics, and in particular of why it is difficult.

5.2 Interrupt handling

In Multics, a system interrupt is one directed to a processor, rather than to a particular process. A typical interrupt is a signal from an I/O controller to a processor signifying completion of some I/O activity.

One way to view the interrupt handling (although not the Multics view[10]) is to see the interrupt as a short term suspension of the executing process. When an interrupt is received the processor resources are deallocated from the process and allocated to handling the interrupt.

What are the process's needs for processor variables at the moment of interruption? The process may be executing any of the instruction programs wired into the processor. Interrupt handling normally does not modify the instructions. The process may need any or all of the

various machine registers, which the instructions use, or the control unit information, used by the lower levels of the system. (An anomaly of the GE645 design is that it is necessary to be able to interrupt the processor during the execution of an instruction.) What the process needs is the precise value of each data variable. On interrupts, therefore, the values of registers and the control unit are copied and stored away. Within a short time the process resumes execution and replaces the values of the processor data variables.

This example of suspension shows how precise delimitation of what resources may be modified during "suspension" facilitates the design of the "suspension" capability. (It also departs from the Multics concept of process in the interest of illuminating the notion of suspension. In Multics a process is closely identified with an address space, and processor interrupts are handled in the address space of the executing process. Hence in Multics the process is not considered to be suspended: only the locus of control has changed.)

5.3 Process swapping

Two further, yet still simple, examples of suspension are provided by the mechanisms for process swapping. When a Multics process is incapable of proceeding because of some needed input, it relinquishes its processor to some

other process. Usually that does not require unloading the process. We consider first the case of processor swapping, then the less frequent case of unloading.

Processor swapping is distinct from interrupt handling in two respects. First, the number of variables deallocated is greater. An example is the associative memory in which some page and segment descriptors are stored for quick reference. Interrupts are handled in the same address space as that in which the interrupted program runs. Both the interrupt handlers and the interrupted programs use the same associative memory, but it is not necessary to store the associative memory when an interrupt occurs because, although its values will change, no incorrect information will be put in the associative memory. But when a distinct process with its distinct memory space takes over the processor, the associative memory will record page and segment descriptors for the distinct memory space. This information is worse than useless for the interrupted process. It is undesirable to record the value of the associative memory (control may remain away from the process for a long enough time that the information becomes invalid) so instead the associative memory is cleared when the process resumes.

In this case the variable being relinquished is one whose content is needed. As its value changes, the process does not want to use the old value, but it also

does not want an incorrect value. Since "undefined", or "empty", is a possible value for any item in the associative memory, the process uses that value on resumption.

When a process has been running for some time it may be "suspended" for a longer period of time. When this happens the process gives up core memory which is allocated to it. In Multics terminology, the process is unloaded.

During execution the Multics system, by dynamic binding, introduces certain dependencies into the process in addition to dependencies implied by the system lattice. For example, Page Control is dependent on a portion of core memory containing the process definitions segment. Page Control can only operate if this block of core is "latched down", i.e., allocated to the process. In addition, the execution of most instructions requires the presence in core of a descriptor segment, which must be in a precise block of core indicated by the processor's descriptor base register. When a process is unloaded the core memory allocated to the process is released and may be modified. When the process is resumed it must be reloaded, i.e., a certain amount of core memory must be allocated to it, including the particular block specified by the descriptor base register. In this example the core memory allocated to the process is a variable which must be reallocated when the process is resumed.

5.4 User faults

Faults in Multics may be system faults, for which standard, mandatory system handlers are provided, or user faults, for which the user may specify his own fault handlers. In the latter category are divide check and overflow faults, among others. The possibility of user-provided fault handlers illustrates a situation which can occur within a single process in Multics, but which presents the problems of suspension. Suppose a user fault occurs for which the user has provided a fault handler. The Fault Interceptor Module of Multics acts as a dispatcher in this case. There is no guarantee that the user's fault handler will ever return; therefore the resources allocated to the process prior to the fault should be deallocated, so that other processes are not adversely affected. Suppose that fault handler runs for a long time--even a week--and then attempts to return control to the point at which the fault occurred. The attempt presents the same problems as resumption of a suspended process.

Any transfer of control to a dispatcher causes the same situation: execution takes an unprogrammed change in direction, from which return within a short or even finite period of time cannot be guaranteed. A similar situation, although one which is programmed, occurs when any program invokes an unreliable program or one which is E-reliable, where E is a longer time than acceptable for

a resource to be allocated to the process.

In these cases, as in the case of suspension of the process for a long period of time, any resource needed by the process may be modified. If the process's dependence on hardware instruction programs, for example, or on other programs is understood, it becomes possible to calculate whether the suspended process or piece of process (before a user fault, for example) can be resumed.

5.5 Complete suspension

We now consider suspension of a Multics process for, say, a week. Such suspension, as might occur when a user runs out of funds or the system undergoes emergency shutdown, is the true subject of this thesis, because it requires that all, and not just a portion, of the process's resources be released. Implementing a suspension capability in Multics has proven to be difficult. The remainder of this chapter is concerned with exploring the nature of the difficulties.

We begin by reviewing the requirements for a suspension capability presented in Chapter One. A system has a suspension capability if it can halt the execution of a user's process in such a way that

(1) other processes are not affected.

(2) some suspended processes can be resumed; namely, those unaffected by any changes which occurred

during the period of suspension.

(3) the system can decide whether resumption is possible for any given suspended process at any given time.

(4) system policy is formulated to guarantee a high probability (near unity) that a process may be resumed at any time within a given T time units after suspension, despite changes to the system.

5.6 Deallocation of resources

The first requirement for a suspension capability is the ability to stop the process without adversely affecting other processes. This means that other processes must not expect control signals from this process and that this process must release all resources allocated to it that might be needed by other processes. We will not discuss control signals here, but instead consider the allocation of resources in Multics.

A fundamental feature of the Multics system is that read and write capabilities (in the sense of permissions--see Chapter Two) are not in general required for reading and writing. That is, two processes may modify a variable simultaneously and inconsistently, or one process may read a variable that is in an inconsistent state because another process is performing a state transition on the variable.

This feature of Multics is not, be it noted, an essential one, but merely reflects the state of the art of multiprocessing in the mid-1960's. It would be possible to implement a revised Multics that enforced attachment and assignment of variables for reading and writing. Dennis[16] and Van Horn[2] suggest machine designs that enforce allocation as a prerequisite for use of resources. The present discussion therefore must be regarded as applying only to Multics in its current implementation.

Because locking (attachment and assignment) is not implemented in Multics processor instructions, it is not enforced for all programs, since the instruction level of the system is the highest level that is used by all higher levels. The Multics system attempts to provide voluntary locking facilities for higher levels. Ilock in the hardcore ring and the Locker in outer rings implement locking conventions. We consider only the locker, since ilock operates in the hardcore ring, which is never current at suspension time.

The locker can make no assumptions about either the programs which use it or the variables which it locks. In particular, it cannot define what constitutes a variable and what does not. That task is left up to the programs that call the locker. But the locker can only be successful if the programs are coded to cooperate in use of the variable, an assumption that should be avoided in the

interest of programming generality. For example, a certain segment might be used by two editor programs and several programs that extract information from the segment for their own use. Some of the latter might be interested in tables or lists within the segment, and define lock words scattered through the segment. The editors should not be required to know about such random locks, nor to cooperate with each other. In a system such as that described by Dennis[16] the hardware defines what may constitute a variable and also provides a locking convention for use of the variables: thus attachment and assignment conventions are enforced for all levels of system.

What are the implications of this feature of current Multics? From our point of view, it means that Multics doesn't fit the system model of Chapter Two. Reading and writing go on independently of allocation. One of the elements needed to compute resumability is a list of resources needed by the process. But just because there is no allocation of segment resources in Multics, there is no information available about what resources the process is actually using.

The most serious consequence is that since there is no information about what variables are being modified there is no possibility of putting those variables into a consistent state. Therefore the system cannot guarantee that suspending a process will not adversely affect other

processes. Various palliatives have been suggested to overcome the basic difficulty for reliable (hence: system) programs; but no complete solution can be found, since the problem stems from a basic feature of the system. The current implementation of Multics cannot satisfy the first requirement for a suspension capability.

5.7 Resuming execution of a process

The second requirement for a suspension capability is the ability to resume a process, assuming resumption is safe. Multics is designed so that a process can be suspended indefinitely and later resumed. Procedures can be constructed for tucking away the per-process information, then later recreating an active process which can be caused to execute. The only mechanical difficulties which exist have to do with repositioning resources needed by the process. Current procedures for mounting tapes, for example, call for the user to telephone an operator and request a tape to be mounted. When automatic positioning of tapes is available, the system should include programs for discovering the desired position for the tape and repositioning the tape. Such repositioning is only possible if I/O system design permits relevant positioning information to be obtained in a straightforward way.

It should be noted that repositioning is in general greatly simplified in Multics by demand paging of most

programs and data. Thus to resume execution of a process it is only necessary to add an entry to the process table. The process will automatically be moved forward in the execution queue until it becomes eligible to execute. At that time a few blocks of core will be allocated to the process by the normal mechanism and when it starts to execute it will allocate core to itself as needed. Thus the module in charge of resumption does not need to concern itself with either processor or core allocation.

5.8 Computing resumability

How can the system decide whether a given process can be resumed at a given time? It must compare the resource needs of the process with available resources. Here we will consider only the most common (and most frequently modified) type of resource in Multics: information contained in segments. Since Multics activates segments on demand, one possibility for computing resumability is to start up the suspended process and let it run until it attempts to activate a segment which it needs and discovers that the segment has been modified or replaced. If this happens the decision is made that the process can not be resumed; otherwise it can be resumed. The advantage of such a strategy is that it uses already existing mechanisms and avoids a specially contrived interface between resumption and Segment Control. However, some special

machinery must be created to indicate that not just any segment with path name X will do. Unique segment identifiers may be used to distinguish between "versions" of a module, viz., between a module and its replacement (distinct interfaces). A retrieval mechanism can then be used to fetch the "safe" version of a segment from the backup system.

There are three disadvantages to the strategy. First, the type of need for each segment is not recorded in the Known Segment Table. Indeed it is buried andor scattered throughout the process's data bases. Hence the system cannot easily know whether a segment's content or interfaces are needed. To find out would require complex and undesirable interfaces between segment control (which is activating needed segments) and various other pieces of system. Hence we must assume that the type of need is unknown and unknowable, and, except for a few special case hardcore segments, assume that the process cannot be resumed (i.e., must be forthwith aborted) if any needed segment has been modified at all. This simplification unnecessarily reduces the probability of resumption.

A second difficulty is that it is only possible to ascertain in Multics when a segment has been modified, not when some small variable in the segment has been modified. A Multics segment can theoretically be very small or very large. But there is no provision for segments within

segments. Further, reasons of cost and efficiency encourage segments to be large (page size is 1024 words, hence a 1-word segment costs 1024 words of core). These two facts combine to mean that small variables are usually placed in large segments. If any word of the segment changes, one must assume that the desired word might have changed, and abort the resumption of the process.

A third difficulty is that some, perhaps most, segments in the Known Segment Table are not needed by the process. Yet because process needs are not known exactly, and also for reasons discussed below in section 5.10, the unneeded segments cannot be purged from the Known Segment Table. Hence if the process later needs a program which it once needed, the system must attempt to get the same version. If the same version is not available, the process is (perhaps unnecessarily) aborted.

In summary, the Known Segment Table is an approximation to the process's abstract (as opposed to physical or hardware) needs. The approximation provides a way (the only way in the present incarnation of Multics) to compute the resumability of the process. It can be used to ensure that a process is not unsafely resumed, but in that case it is practically guaranteed to prevent the resumption of some processes unnecessarily. Hence by satisfying the third condition for a suspension capability, it works against the fourth, a high probability of resumption.

5.9 Probability of resumption

The fourth requirement for a suspension capability is a near-unity probability that a suspended process can be resumed within some advertised and reasonable T time units after suspension, despite changes in the system. Chapters Three and Four discuss ways of changing the system without axing suspended processes. The previous section of the present chapter showed how one feature of Multics practically assures a non-unity probability of resumption whenever changes are made to user or system resources in the process's address space, even when the process could safely resume execution despite the changes.

We now apply the results of Chapter Four to a consideration of the Multics distributed supervisor, to see how the system supports resumption when challenged by a distributed supervisor. (To avoid any possible misunderstanding, I wish to point out that the strategy to avoid awkward suspension described below is not mine.)

In Multics the supervisor appears in every process. This arrangement is referred to as a distributed supervisor[11]. Its motivating strategy is to allow each user process to supply its own needs, as opposed to having user processes place requests in the queues of supervisor processes. One consequence of the strategy is to make mandatory close cooperation among processes, and therefore strong interdependencies. We examine the consequences of

suspending a process while some program of the supervisor is current, and then attempting to resume the process after modifications have been made to the supervisor. The examination is not intended as an exhaustive treatment but rather to indicate the nature of the difficulties involved in suspension.

Suppose that a process is suspended when some supervisor program is current, and that while the process is suspended that program is modified. We assume that the new program cannot be transplanted into the process. Can the process resume with the old version of that program? To rephrase the question: can two processes have different versions of a supervisor program? There are several reasons for avoiding such a situation.

First, the success of some algorithms used in Multics requires that each process use the same algorithm. Two examples are page control and the locking mechanism. It is difficult to ensure that two processes are supporting a single paging algorithm if each process uses a different paging program. Similarly, the locking strategy in Multics is such that each process relies on other processes to inform it that needed data variables have been unlocked.

Second, even when different algorithms might meaningfully be used by different processes, the interfaces between processes must be constant. For example, the list-dir primitive of directory control lists the contents

of a directory. One process might reasonably list the contents in a different order from another process; yet both must know the correct structure of the directory. Similarly segment control and the traffic controller in each process communicate by means of common tables, and all must share the same declaration for the structure of the tables.

A third consideration renders suspension in the hardcore supervisor unwise in the current implementation of Multics. While processes share the text of programs, certain other per-program information such as the linkage section is unique to each process. For reasons of efficiency all processes share common linkage sections for supervisor segments. This greatly reduces the core requirements of the supervisor, but increases the interdependence of processes. Two processes may have different versions of a program only if the versions have identical linkage sections.

For these reasons it is unwise, although theoretically possible, to replace modules of the distributed supervisor. With luck and fastidious bookkeeping, a substitution is possible. That is, if a process has been suspended while a program of the distributed supervisor is current, then it can theoretically be resumed.

Since the system can guarantee, however, that all hardcore programs are reliable (remember we assume no program bugs!), it can ensure that all processes have the

same versions of hardcore programs by not allowing suspension in the supervisor. This decision has been implemented in Multics by not allowing suspension in the hardcore ring. It should be noted, however, that similar considerations apply to any programs which rely on or cause close interdependence between processes, for example accounting programs using common account data bases or programs for interconsole communication. The simple way to avoid insoluble transplant problems involving these programs is to insist on their reliability and to forbid suspension when they are current.

We conclude by remarking that even a cursory examination of the Multics distributed supervisor has illuminated some problems inherent in the suspension of closely cooperating processes.

5.10 Dynamic linking

One way to decrease the probability of resumption of a suspended process is through binding. Binding a program to a set of machine instructions (compilation) creates dependencies as reflected in the system lattice. Dynamic binding creates dependencies not reflected in the system lattice. If the binding is interpretive, no new needs are introduced into the process. An example is the interpretation of core addresses in Multics. If a page is located one time at this address, one time at

that, the interpretive paging hardware operates to find the correct absolute address, and the process does not need the particular block of memory where the page was first located. Corbató and Saltzer[3] have referred to this interpretive binding as reversibility of binding. We now consider an example of dynamic binding in Multics which is not reversible, and which therefore introduces new needs for the process and lowers the probability of resumption.

As the grand finale to this chapter, we explore the consequences for suspension of one feature of Multics, intersegment linking. Every segment residing in the Multics file system can be referred to by a pathname designating its place in the hierarchy. A pathname, however, can be a very long string of letters, and leads to inefficiency in intersegment references if it is used as the means of addressing segments. For this reason the address space of a Multics process is not the file hierarchy, but a vector of segments. In the first reference of a process to any segment X, the segment is assigned a place in the vector, i.e., a segment number. Thereafter all references to X must be made by segment number. One way to make such references would be for the process to compute the segment number at every reference to X. That, however, is a time-consuming procedure. It must be done each time a new segment references X, but later references

to X by the same segment can be expedited if the segment retains a copy of X's segment number.

To accomplish this, the system makes pointers, or segment+offset addresses, available to referencing programs. The linker places pointers in a segment's linkage section, and system programs freely copy pointers into their stack frames and static data areas.

That strategy promotes efficiency, but it also introduces a dependency in the process that is not inherent in the system lattice. Namely, the segment using the pointer is now dependent on the association of X and a particular number. That is, the compound of the pointer and the entry in the Known Segment Table which defines the association becomes a static private variable on which the process is dependent.

One result of the dependency is to make termination of segments, that is, dissociation of a number and a segment, extremely dangerous. Within the process the supervisor should ensure that no segment is terminated while any module is still dependent on the association of a pointer and the Known Segment Table entry for that segment (as there is no mechanism in current Multics for tracing references to a segment number, this implies that no segment can be terminated). Further, the Known Segment Table entry must be regarded as attached to the process, so that the process is not resumable if the entry is modified.

(Actually the essential element of the entry is the association of segment and number. Any transformation of the Known Segment Table which preserved that association would not render suspension impossible. Such a transformation is in theory possible because the process is dependent just on the contents of the entry--see Chapter Three.)

Multics intersegment linkage makes a referencing program Y dependent not only on the segment number of X, but also on the entry points of X. That is, Y's linkage section records for X and a symbolic entry name (i.e., any location which can be symbolically referenced from other segments) a numerical offset within X. As a result, the process is dependent on the association of a symbolically referenceable name in X and the word number to which the name corresponds. This association is recorded in the linkage section for X when X is assembled. After the linkage is made, Y cannot execute correctly if that association of internal name and word number is changed.

Thus intersegment linkage creates dependencies in a process which are not implied by the system lattice. Not only are current programs and static data variables needed but also, if in the future execution of the process any reference is made to a previously-known segment, the process must use some version or replacement of the segment that satisfies very strict interface constraints.

The result is twofold: statement of minimal requirements for resumption is made more complicated, and the possibility of resumption is reduced as the needs of the process are increased.

The Multics strategy of intersegment linkage may be contrasted with a hypothetical system proposed by Van Horn[2] and a practical proposal by Dennis[16]. Each of these advocates abolishing the distinction between file system organization and process address space. Van Horn proposes addressing by segment numbers to the exclusion of pathnames, while Dennis advocates, and apparently intends to implement, processing hardware that allows instructions to address hierarchical memory without segment numbers. Either of these schemes eliminates the need for Multics-type intersegment linkage and the resulting gratuitous dependencies created in processes.

CHAPTER SIX

CONCLUDING OBSERVATIONS

The few general purpose computing systems which have suspension capabilities have demonstrated the usefulness of such a capability, a usefulness which is practically equivalent to a need. M.I.T.'s CTSS[17] has such a capability, and I have heard people make such remarks in conversation as, "CTSS became really useful once it started saving my job when I was thrown off."

A suspension capability, then, would seem to be a requirement in future systems. But it cannot be added as an afterthought. Careful design is necessary, especially in a system built to evolve, if the suspension module is not to be changed as often as the system, or indeed, if the system is to have a suspension capability at all.

This thesis has been an attempt to contribute to the design of systems simply by setting down in writing the need for a suspension capability, and defining its constituent capabilities. I have attempted to show some implications of a suspension capability for systems in which information, and not just copies of information, is shared.

The thesis has focused attention primarily on two related requisites for a suspension capability: to know a process's needs so that it is not incorrectly resumed

(i.e., allowed to resume using an incorrect replacement for some needed resource); and to maximize the probability of resumption of a process within some acceptable period after suspension, despite system changes.

The latter requirement has been called by Corbató and Saltzer[3] the need for "reversibility of binding". We conclude with a few remarks on reversibility of binding. Binding is "an operation which occurs at a variety of levels in a computer system: the choosing of a particular hardware and supervisor environment in which to implement a program construct."[3] On inspection, it seems reasonable to consider binding in three distinct categories: program to program binding, as in compilation of a FORTRAN program in terms of machine instructions and other programs; program to data binding, as when a program includes table lookup; and information to machine binding. The last occurs when physical pieces of hardware are chosen for representation or execution of information. Examples are loading registers in core memory with bit patterns of information, or causing a processor to execute a stored program.

The 1960's have seen a recognition of the need for the reversibility of the third kind of binding. Segmentation, core memory management, and file system management of secondary storage in Multics, for example, all enable a process's need for a particular unit of hardware to be ignored as soon as the process deallocates the unit: the

process may be resumed later with other, interchangeable pieces of hardware. Thus the process is made independent of a particular hardware configuration.

Information-to-machine binding is reversible in the sense that identical hardware units may be used interchangeably. The first two kinds of binding are of a different nature, since what is bound to is abstract and easily modified. We say that binding is "reversible" in these cases (information-to-information binding) if the module to which something is bound can be replaced by a module of different content or, more demanding, by a module with different interfaces as well. Thus the requirements of reversibility of binding are stronger for software than for hardware.

The system model presented in Chapter Two reflects the static information-to-information binding in the system. As the system runs, dynamic binding takes place, introducing further dependencies within the context of a single process. Sometimes this is also desirable or necessary, as when the choice of a particular square root routine is left until execution of the process. The Multics dynamic linking strategy introduces another kind of dynamic binding, in this case undesirable, as shown in Chapter Five. To facilitate reversibility of binding and hence the design of a suspension capability, systems should be designed so that dynamically introduced bindings are reversible or, where not

explicitly desired, avoided altogether.

REFERENCES

1. Dennis, J. B., and E. Van Horn,"Programming Semantics for Multiprogrammed Computations," Communications of the ACM 9 (March, 1966), pp. 143-155.

2. Van Horn, E., "Computer Design for Asynchronously Reproducible Multiprocessing," Ph.D. Thesis, M.I.T. Department of Electrical Engineering, September, 1966.

3. Corbató, F.J., and J. H. Saltzer, "Some Considerations of Supervisor Program Design for Multiplexed Computer Systems," IFIP Congress 1968 Invited Papers, North-Holland Publishing Company (Amsterdam), pp. 66-71.

4. Corbató, F. J., and V. A. Vyssotsky, "Introduction and Overview of the Multics System," AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 185-196.

5. Glaser, E. L., et al., "System Design of a Computer for Time Sharing Application," AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 197-202.

6. Vyssotsky, V. A. , et al., "Structure of the Multics Supervisor," AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 203-212.

7. Daley, R. C., and P. G. Neumann, "A General-Purpose File System for Secondary Storage," AFIPS Conference

Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 213-229.

8. Ossanna, J. F., et al., "Communication and Input/Output Switching in a Multiplex Computing System," AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 231-241.
9. David, E. E., Jr., and R. M. Fano, "Some Thoughts about the Social Implications of Accessible Computing," AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 243-247.
10. Saltzer, J. H., "Traffic Control in a Multiplexed Computer System," Sc.D. Thesis, M.I.T. Department of Electrical Engineering, June, 1966.
11. Bensoussan, A. , C. T. Clingen, and R. C. Daley, "The Multics Virtual Memory," paper presented at the Second ACM Symposium on Operating System Principles, October, 1969.
12. Dijkstra, E. W., "Structure of the THE Multiprogramming System," Communications of the ACM 11 (May, 1968), pp. 341-346.
13. Dijkstra, E. W., "Complexity Controlled by Hierarchical Ordering of Function and Variability," Software Engineering, report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, published 1969, pp. 181-185.
14. Parnas, D., "More on Simulation Languages and Design

Methodology for Computer Systems," AFIPS Conference Proceedings 34 (1969 SJCC), Spartan Books, Washington, D. C., 1967, pp. 739-743.

15. Randall, B., and F. W. Zurcher, "Multi-level Modelling--A Methodology for Computer System Design," IFIP Congress 1968, North-Holland Publishing Company (Amsterdam), pp. D138-D142.
16. Dennis, J. B., "Programming Generality, Parallelism and Computer Design," M.I.T. Computation Structures Group Memo #32.
17. Crisman, P. A. (editor), The Compatible Time-Sharing System: A Programmer's Guide, second edition, M.I.T. Press, Cambridge, 1965.