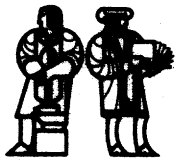
LABORATORY FOR
COMPUTER SCIENCE
(formerly Project MAC)



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

14

MIT/LCS/TR-175

A ROBUST ENVIRONMENT FOR 30 PROGRAM DEVELOPMENT

Harold J. Goldberg 18

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Cite as 8½"

This blank page was inserted to preserve pagination.

MIT/LCS/TR-175

A ROBUST ENVIRONMENT FOR PROGRAM DEVELOPMENT

HAROLD JEFFREY GOLDBERG

February 1977

This research was performed in the Computer Systems Research Division of the M.I.T. Laboratory for Computer Science. It was sponsored in part by Honeywell Information Systems Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641, which was monitored by ISTAO under contract No. F19628-74-C-0193.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

A ROBUST ENVIRONMENT FOR PROGRAM DEVELOPMENT *

by

Harold Jeffrey Goldberg

ABSTRACT

This thesis examines the problems of debugging and preservation of the user programming environment and proposes a scheme by which the program development environment can be protected.

Typically, designers of timeshared or multiprogrammed computer systems only consider inter-user interference as a source of problems and do not worry about what users do in their own environments. Thus, users can, by writing incorrect programs, cause the destruction of the programming environment and personal data bases. A protection scheme is proposed that satisfies the needs of the user by employing a protection mechanism, rings, that allows the program development environment to be protected from user written programs and yet be outside of the supervisor. Having these programs outside the supervisor satisfies the goals of creating a "security kernel", which is a supervisor containing only security related programs.

The thesis presents a model of the user environment wherein the concept of a "procedural package" is explained. The procedural package contains not only the code for the procedure, but in addition, environment components necessary for the proper execution of the procedure such as dynamic, static, and allocate/free storage. The thesis describes the "inter-procedure interference" problem in terms of the model and proposes an ideal solution based on a domain architecture. Problems with the ideal solution are presented and an alternate solution suggested.

In addition, the thesis identifies and discusses, in detail, environments that are needed to control a user's process, and examines error signalling mechanisms, particularly in their use in an environment like the one proposed to solve the inter-procedure interference problem.

THESIS SUPERVISOR: David D. Clark
TITLE: Research Associate of Electrical Engineering and Computer Science

---

* This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on January 21, 1977 in partial fulfillment of the requirements for the Degree of Master of Science.

ACKNOWLEDGMENTS

I first want to apologize, in advance, to all those that I may have forgotten to mention in this section, but richly deserve to be included.

I would like to thank...

...Dave Clark, my thesis supervisor, for his guidance from conception of this research through writing of this thesis, for his help in defining and discussing the issues and providing structure and organization to the ideas presented in this thesis, and finally, for his patience and quick turnaround time in reading drafts.

...Jerry Saltzer for reading and commenting on a draft of this thesis (and correcting my english).

...Dave Reed for discussing topics in this thesis, especially signalling, and discussing topics not in this thesis but that attempted to keep me a well-rounded person.

...Doug Wells for discussing some of the issues in this thesis and answering many implementation related questions about Multics and the "right way" of programming.

...All members of CSR, especially Art Benjamin, Nancy Federman, Harry Forsdick, Doug Hunt, Steve Kent, Al Luniewski, Warren Montgomery, Karen Sollins, and those already mentioned above for making life at CSR and M.I.T. an enjoyable and rewarding experience.

This section would not be complete without special thanks to my dear wife Esther, without whom this thesis, nor my graduate student career at M.I.T., could ever have have been completed, and to my parents, brothers, in-laws, and friends for their moral support and encouragement throughout two and one half long years of graduate study.

This research was performed in the Computer Systems Research Division of the M.I.T. Laboratory for Computer Science. It was sponsored in part by Honeywell Information Systems Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641, which was monitored by ISTAO under contract No. F19628-74-C-0193.

To Esther,

who makes all the difference.

TABLE OF CONTENTS

ABSTRACT 3

ACKNOWLEDGMENTS 4

DEDICATION 5

TABLE OF CONTENTS 6

LIST OF FIGURES 8

1. Introduction 9

1.1 Motivation 10
1.2 Plan of the Thesis 12
1.3 Related Work 14

2. The Process Model and the User Domain 17

2.1 Supervisor and User Domains 18
2.2 Dissecting the User Domain 19
2.3 The Problem 22
2.4 An Ideal Solution 25

3. A New Model 29

3.1 Functional Components in the New Model 29
3.2 Deciding on a Protection Mechanism 33

4. A Closer Look at the Support Routines 39

4.1 Two Cases of Support Routines 39
4.2 Guidelines for Support Routine Coding 40
4.3 Examples 42

5. Command and Control Routines 49

5.1 Structure of Environments 49
5.1.1 Command Environment 51
5.1.2 Control Environment 53
5.2 Processing Commands 54
5.2.1 Text Substitution 56
5.2.2 Parameter Evaluation 56
5.2.3 Command Files 59
5.2.4 The Command Processor 60
5.3 Getting to the Control Environment 61
5.4 Implementation of Command and Control Environment 63

5.5 Protecting the Command and Control Environments . 63
5.6 Design Decisions Based on the Protection Scheme . 67
5.6.1 Command Processing Revisited 67
5.6.2 Command Processor Escape Mechanism 70

6. Signalling 73

6.1 Purpose of Signalling 74
6.2 A Model for Signalling 74
6.3 Extensions to Signalling 76
6.4 Signalling Problems 77
6.5 Multi-Ring Signalling 82

7. Implementation, Conclusions, and Future Research 87

7.1 Implementation 87
7.2 Conclusions 92
7.3 Areas for Future Research 93

Appendix A. Certification and Kernel Simplification 95

BIBLIOGRAPHY 97

LIST OF FIGURES

Figure 2.1 Supervisor/User Domains 19
Figure 2.2 Protected Procedure Package 26
Figure 3.1 Functional Boundaries 30
Figure 3.2 Proposed Structure 35
Figure 5.1 Command and Control Structure 50
Figure 5.2 Abbreviation Processing 57
Figure 5.3 Total Command Processing 58
Figure 6.1 Sample Call Stack 79

Chapter One

Introduction

The need for increased efficiency in the use of large systems led to the development of multiprogrammed systems. This led the way to the development of interactive timesharing systems. Of major concern in the design of these systems was the separation of users so that one user could not affect the programs, data, or operation of others and yet users could still be allowed controlled sharing. This concern evolved a frame of mind which based system design decisions on whether or not a user, by any action, could affect the system or another user. Self-annihilation (of programs) was considered legal and permissible under these rules as long as this could not affect the system security.

It is this "laissez faire" attitude towards user programs that is dealt with in this thesis. Systems should provide some self-protection mechanisms to help users detect and limit the scope of errors, thus speeding up the development of programs. This thesis describes how to accomplish this goal by employing a protection mechanism (rings) to protect a "program execution environment", containing program support functions, from user-written programs.

1.1 Motivation

As hardware costs come down, the cost of software begins to dominate the cost of a computer based product. Helping the programmer would therefore reduce software expense and project cost. Numerous methods of helping a programmer write a correct program the first time have come about and/or are being studied including high-level languages, structured programming, module specification, and a programmer's apprentice (see "Related Work" section). In the long run, current research might pay off; but until that happens many programmers will face the problem of debugging their software in the "old fashioned" way.

The proposed program execution environment will help programmers catch errors during testing and limit the scope of those errors. This capability will undoubtedly speed up program development and thus reduce software cost.

Unfortunately, programs are never completely bug free even after years of development. Constant upgrading and changing patterns of use may exercise portions of a program that were not considered important and therefore not thoroughly tested at the time of the first writing.

To help in these situations a protected program execution environment can be employed that would be safe from user program modification and could notify the user immediately upon detection of an

error instead of letting the error propagate and cause more extensive damage. This monitoring will be referred to as "guarded execution".

Program development and guarded execution justify the work done in this thesis, in general. However, the actual motivating force that led to this research was the Multics kernel design project being carried on in the Computer Systems Research division of Project MAC at M.I.T. (presently the M.I.T. Laboratory for Computer Science) [Schroeder 75]. Appendix A contains a brief discussion of that work and should be reviewed if the reader is unfamiliar with the concepts. In summary, certification of correctness of the security features of the supervisor would be easier if the supervisor were made smaller. Modules unrelated to security would be removed from the supervisor and placed in the user domain. This increase in the number of support modules existing in the user domain increases the fragility of the user domain and the possible scope of damage that errors can cause there. Thus program development becomes a more difficult task.

The combination of the requirements of certification and the traditional "don't care" attitude of the systems developers (described in the introduction) resulted in the research reported in this thesis.

1.2 Plan of the Thesis

The thesis proper begins in chapter 2 with the description of a basic model of a process in a multiprogrammed timesharing system, in which Supervisor/User modes are briefly discussed. The User domain is then dissected into procedural packages containing basic environment components (as opposed to functions) and the problems of access by the wrong procedure to those components are discussed. Efficiency considerations that led to the coalescing of distinct components into a single object on a particular system are discussed and studied. How this coalescing aggravates the problems of incorrect access to environment components is explained.

An ideal solution to the inter-procedure interference problem is proposed wherein each procedure would exist in its own protection domain and could not affect any of the environment components of another procedure. Some basic problems of complexity and efficiency of using this approach are discussed.

A new model of the user domain based on functional units is then described in chapter 3. The concept of support routines and command and control routines are explained. A new, simpler solution to the inter-procedure interference problem is proposed in which the support routines are protected as a unit. A mechanism for implementing this new proposal is then discussed.

Chapter 4 takes a closer look at the support routines and offers suggestions for their proper coding so that their protection is facilitated. Examples of major modules that need not exist in the supervisor and could benefit from protection in the user domain are discussed.

In chapter 5 the programs that a user controls a process with, the command and control routines, are studied in detail. Command line processing and process control are discussed. The reason for protecting these routines is explained and possible methods of protection are considered, including a separate process "front end" approach.

Chapter 6 discusses error signalling mechanisms and their problems. These problems can be ignored in a single domain implementation but must be considered in order for a multi-domain environment (such as the one proposed in this thesis) to work.

Finally, chapter 7 presents conclusions, a summary of results, and comments about a test implementation. Possible areas for future research are also discussed.

Appendix A contains a detailed discussion of certification and explains why reducing the size of a system supervisor facilitates verifying its correct operation.

1.3 Related Work

This research was motivated by the kernel design work in the Computer Systems Research division of Project MAC at MIT (now the Laboratory for Computer Science). [Schroeder 75] discusses the CSR kernel design project and the method of attack. [Janson 74] and [Bratt 75] describe how programs responsible for dynamic linking and namespace management, respectively, can be removed from, and thus simplify the supervisor. [Montgomery 76b] discusses how process creation can be partially removed from the supervisor thus aiding in the simplification goals.

Many researchers are investigating techniques to prove programs correct and/or help users write correct programs. [Parnas 72a] describes module specification techniques. [Liskov 76] is developing a language, CLU, which is thoroughly type-checked with hopes that it will prevent (or reduce) programming errors and allow automatic program verification. It allows the creation of extensible objects which are manipulated by type managers or "CLUsters" and prevents all other programs but the appropriate type managers from directly touching the "insides" of the extended type object. [Hewitt] describes a programmer's apprentice which should help a program writer avoid such errors as incorrect number of parameters to subroutines and wrong types of arguments, thus maintaining consistency of specifications, and answer the programmer's questions about dependencies between modules.

There are many theses and papers related to the ideal solution presented in chapter 2, which is basically a domain architecture. [Dennis 64] and [Dennis and Van Horn 65] describe spheres of protection. [Schroeder 72] describes the hardware that could support mutually suspicious domains. [Redell 74] discusses type extension and revokable capabilities. [Jones 73] describes capabilties in a very formal sense.

The concepts of command and control environments are discussed in this thesis. The General Electric Mark II time-sharing [Montgomery 76a] is an example of a system which contains these functions in supervisory code which I claim is unnecessary given the proposed protected environment within the user domain. [IBMCP] is an example of a system that contains the control environment, but not the command processor, within the supervisor. [TENEX] contains the command processing functions coded as system calls while the program that accepts user commands is in the user domain. [NSW] and [DCS] are examples of current research that promote the approach of a front end processor.

Of course there is much literature on Multics but I have chosen [Organick 72] for an overall description of concepts although the details have changed somewhat, and [TR123] because of its extensive list of references. [Schroeder and Saltzer 72] describe the ring hardware in use on Multics today.

Debugging and maintaining the user's process is of prime consideration in this thesis. Having a debugger protected from user written code certainly helps in the job of debugging and guarantees that

some portion of the process is preserved despite user programming errors. [IBMCP] has a primitive debugger in supervisory code that is protected. The debugging system, "NURSE", described by Gould [Gould 75] is more extensive than the IBM debugger, but it too is in the supervisor. Yates' thesis [Yates 62] contains a proposal for a protected debugger that would be an "administrative routine", which is also in the supervisor. A better approach is found in [PSN25] and [PSN26] which proposes to protect a debugger in the user domain using user controlled base and bound registers on [CTSS]. This approach is much like the ring structure proposed in this thesis.

Chapter Two

The Process Model and the User Domain

In order to describe more accurately the details of this research, a model of a user's computation is needed. The model chosen was based on the Multics system; however, there are not that many Multics specific items in the model. Those items that are not present in a given system can simply be ignored without loss of applicability. The model is intended to reflect a typical process in a multiprogrammed computer environment.

The term "process" appears throughout this thesis. The association one should make with this word, while reading this thesis, is an active agent, a processor (real or virtual, i.e. multiplexed), following a sequence of instructions. The past, present, and future effects of the instructions executed is a process. The terms "procedure" and "program" are used interchangeably, as are the terms "parameter" and "argument".

2.1 Supervisor and User Domains

First the process is broken down into two scopes of access, or "domains", namely "supervisor" and "user", as shown in figure 2.1. The figure is intended to indicate that the access privileges of the user domain are a subset of the privileges of the supervisor domain. The access privileges of the supervisor include access to I/O channels and devices, which may be controlled by a different means from data access. (1) All interrupts and faults are directed to the supervisor domain so that it may properly control the user domain and I/O. There may be one or more real or virtual (multiplexed) processors executing the process, but for simplicity only a single processor is assumed.

Now the user domain is examined, which is where the problem being attacked exists.

---

(1) For example, Multics uses a privileged state bit in the processor to allow execution of I/O instructions. PDP/11's have no I/O instructions; the device registers are addressed as memory locations.

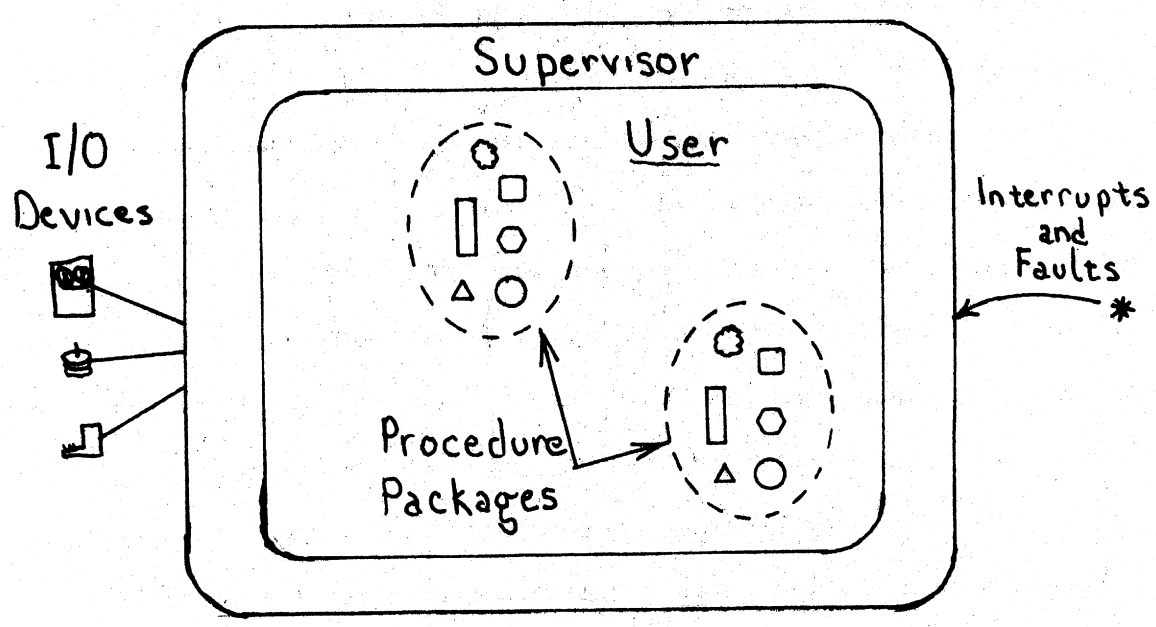


Figure 2.1 Supervisor/User Domains

2.2 Dissecting the User Domain

The user domain is considered to be the collection of procedures that execute there. The code is not the only item that allows a procedure to function, though. The description of the user domain will consist of what environment is "seen" by programs that run in that domain. The user domain is examined in this peculiar way so that the program interference problem can more accurately and precisely be described.

Programs are viewed as "packages" containing the code and additional objects that are needed for execution. This section identifies those objects that make up the environment necessary for most program execution. Much of the environment has been established as conventions, but these conventions are indeed necessary for the proper execution of programs written in higher level languages. Only passive objects are mentioned in this analysis. Functionality that exists because of either user or system programs is simply and grossly categorized as user programs accessible via call, and supervisory programs accessible via gates (an abstract entity for supervisor calls).

The passive objects that this research has identified are small in number, but are intended to capture a particular view of the process which will be examined shortly. The components are 1) dynamic storage, 2) allocate/free storage, 3) "own" storage, 4) parameter list, 5) parameters, 6) linkage, and 7) databases.

The dynamic storage area is where blocks of storage are automatically allocated each time a procedure in called, and automatically freed when the procedure returns. The class of storage is known as "automatic" in PL/I. Temporary work areas are typically allocated in this area. The allocate/free area is where blocks of storage are allocated and freed under direct program control. These allocated blocks remain in use even after the procedure that performed the allocation has returned. A typical use for this type of area is for the allocation of I/O buffers and I/O access method control blocks (e.g.

the index of an ISAM file). "Own" storage, or internal static, is automatically allocated upon first entering a procedure and is not usually released. The allocation may be implicit, since some systems provide this storage class as a physical part of the executing program (impure procedures). A typical use for this type of storage is for saving information between program invocations, such as whether the procedure has initialized a data base, or pointers to blocks of Allocate/Free storage. (2)

The parameter list identifies arguments to the called procedure, and may include descriptions of these arguments including type and length information. The parameters themselves are typically located in some storage external to the called procedure. A linkage section, or transfer vector, is used to bind named objects to physical machine addresses. Finally, the last component of the procedural package is the collection of data bases that are in use by programs. On some systems this component can be considered the open files for the program.

Note that only the parameters and databases are, or need to be, external to the procedural package; the other components either should exist only in the procedural package, or be copied in. In no case should these other components be referenced by external procedures unless explicitly passed as parameters. The violation of this maxim is exactly the problem that this thesis examines in the next section.

---

(2) The compiler does not know the address of Allocate/Free blocks because the programmer may allocate them in any procedure and pass pointers around. The compiler, however, assumes that internal static is allocated in a certain place, possibly pointed to by a register.

2.3 The Problem

The reason for discussing this particular breakdown of the program execution environment is that implementation of the specified components tend to cause unnecessary common mechanisms between procedures in the user domain. For instance, the dynamic storage area is typically implemented as a single object shared between programs - the "stack". Usually the Allocate/Free area is a single area for all of the programs in the user domain to share. Linkage is usually implemented as a single common transfer vector for an entire collection of programs. On systems where programs are kept pure, internal static is not part of the program object module and is allocated in an area with all the other static sections in the user domain. Parameter lists are typically allocated in the dynamic storage area, causing further entaglement of objects. On systems which have virtual memory (segmentation), every known segment is a potential data base. (3) These examples of the concentration of multiple objects into one object which is global, allows one procedure to access another's "piece" of the object. This is possible because access control usually extends only to individual objects and not to sub-objects. Because of this, one faulty procedure can modify and destroy another's dynamic, allocated, or static storage.

---

(3) A known segment is a file of which the address can be constructed. On Multics, the address is a two-dimensional quantity (S,0). "S" represents an index into an array of "open" files, and "0" is the offset into the segment S. Hence an arbitrary value for S can select any of the open, or known, segments.

This data concentration may result in unpredictable results of tested and debugged programs. A process may be totally destroyed because of one modified bit. Programs may reference the wrong block in an allocated/freed, static, or linkage area. File access information can be destroyed, causing jumbling and loss of records. Data bases may be written instead of read. The stack may be overwritten or deleted. Programs and linkage may be written over. For those systems (like Multics) where gates are actual programs found by the normal linking mechanism (as opposed to SVC type instructions), destroying linkage can prevent any further calls to the supervisor.

An example of the merging of unrelated small objects into one large one with no specific access control to prevent erroneous access is found in the current implementation of the Multics stack. In the original Multics design, segments were supposed to contain only data requiring identical access. This was not followed on Multics when the current stack design is examined. Efficiency, related to reduced page faults, caused the merging of unrelated data into one object. One segment was used for all stack frames, and at the low end of the stack the infamous "base of the stack" was designed.

This base of the stack contains many special environment definitions including pointers to important entries in PL/I operators (i.e. call, save, return, and stack manipulations), to the procedure to do the signalling in that ring, and to the Linkage and Internal Static

Offset Tables, (4) to name a few. Again, these objects were all merged simply for increased memory utilization.

Because of the merging of all the above areas into one segment which was writeable, one bad stack reference could destroy the entire execution environment.

Until recently, linkage and internal static sections were always combined into one static section. This allowed out of range references to internal static to affect linkage of programs.

In order to prevent these types of errors from occurring, and possibly to catch them as they occur (for debugging), it would be reasonable to protect these elements of the execution environment which are common mechanisms. There are in existance machines which prevent this sort of wild access from occurring, but are limited to stack references and a few data types [Organick 73]. Current research has shown ways for preventing this type of behavior in general, (5) but they are just not applied because of efficiency considerations. In the next section an ideal solution and the problems with it will be discussed.

---

(4) The Linkage and Internal Static Offset Tables are used by procedures to locate their linkage and internal static areas within the segments containing all linkage and internal static areas.

(5) Interpreters and extended type objects and type managers for example.

2.4 An Ideal Solution

To prevent the type of interprocedure interference that could arise due to sharing of environment components, a protection mechanism of some kind is needed. This protection is required not only for the components of the execution environment discussed in the previous section, but for the actual code of the procedures as well. Ideally, the procedural package concept would be enforced by the system; each procedure would be housed in its own protection domain and be allowed to access only objects that it required for operation. These objects might be catalogued in a list which would quickly identify all those objects that the program in that environment could ever reference. (6) Among those objects would be the components of the execution environment. Only required access to each component would be granted so that, for example, only read access is allowed for linkage sections and only execute access for object segments. Furthermore, the components would not be accessible at all outside the program domain unless explicit permission were given. This structure is depicted in figure 2.2.

There should be ways of adding and removing objects from the "list". These addition and removal operations are necessary so that blocks that are allocated and then freed would not be accessible to the

---

(6) This could lead to an easily implementable system given appropriate hardware.

original allocator once they are freed, and so that parameters could be passed for use, then removed from the "list" when the procedure returned. These parameter capabilities are shown as dotted capabilities in figure 2.2.

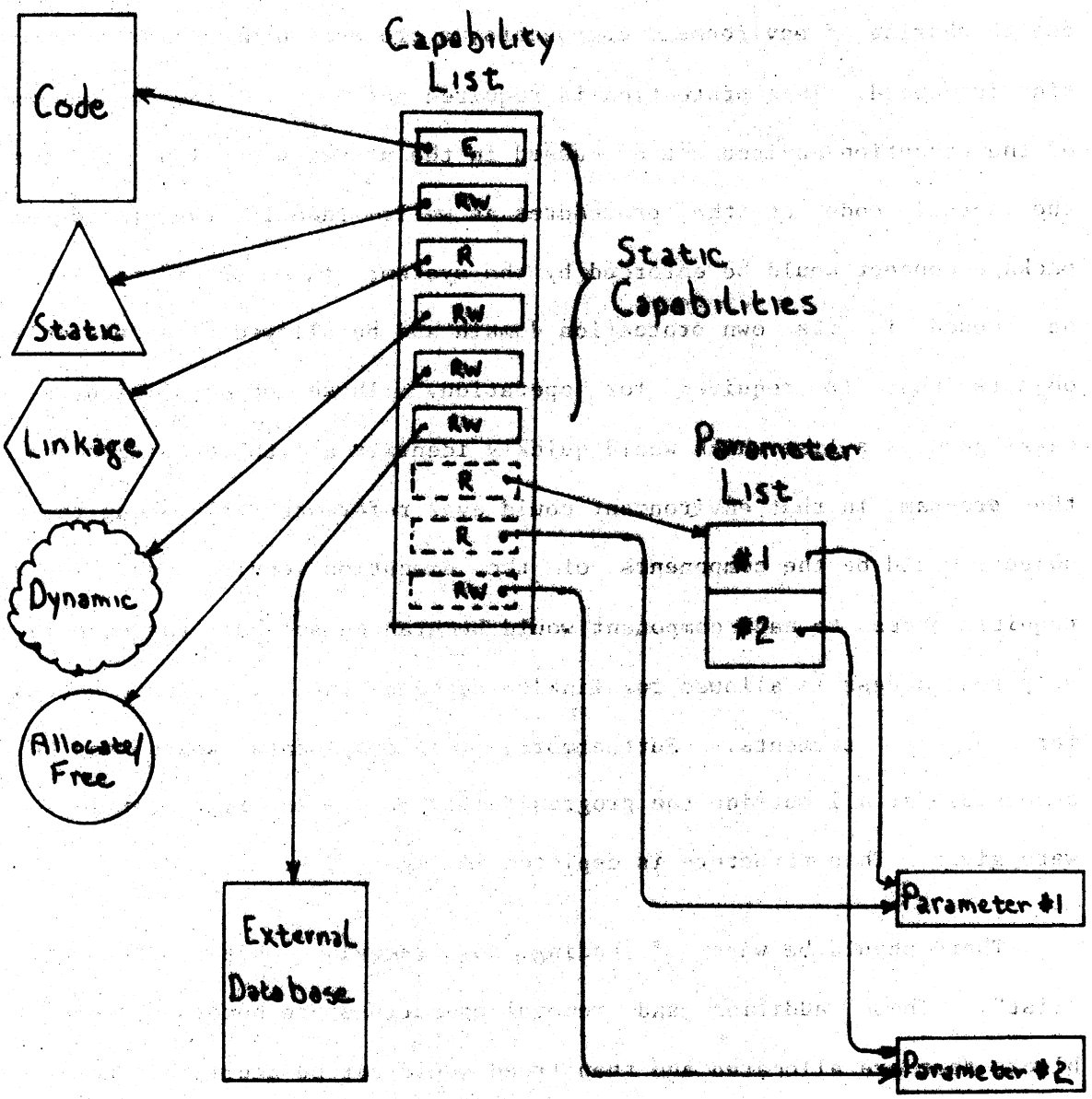


Figure 2.2 Protected Procedure Package

There is some expense incurred when dealing with a system with this architecture, however. Maintenance of the individual capability lists for each domain requires an extra database for each program. With a sufficiently large collection of programs (as on a typical large system), this could involve many such objects. In addition, each environment component would be a distinct object. Thus, the implementation is faced with the management and storage of all these objects that exist as independent entities. (7) During execution, domain changing with every procedure call would undoubtedly be expensive because dynamic capabilities would have to be stacked and maintained [Schroeder 72]. Excessive paging overhead due to a new capability list and environment components would also decrease efficiency. As pointed out earlier in the Multics example, excessive paging overhead is in fact what led to some of the problems being attacked in this thesis. Returning to the original state of affairs just for the promise of better debuggability is probably not sufficient motivation to reduce performance.

Besides performance issues there may still be a problem in using a program per domain system. Examining in detail how a user could create and develop programs points out some problems that have to be overcome if such a system is to be considered useable. In particular, consider the set of privileges that have to be granted when a new program is created. First, the source must be entered and edited, which requires

---

(7) This problem has come to be known as the "small objects" problem.

that an "editor domain" have read and write access to the source. Then a compiler must be able to read the source and write the object segment. A debugger must be able to read and write the object segment (perhaps only temporarily) so that code can be examined and breakpoints set.

All seems fine so far, but it is in the execution of the procedure where the more complex access problems and controls come into view. The new procedure must be given the privilege to call all the subroutines it uses. Similarly, if it is called by another procedure, that other procedure must be given the right to call it. All the support programs (e.g. linker, dynamic area manager, etc.) must be given the appropriate access so that they may operate on the parts of the domain that they are intended to (linkage section, dynamic storage, etc.). Clearly, if users had to establish all of these privileges manually, they simply would not do it; and what good is a protection mechanism if it is not used?

To help in this problem one might decide that a group of programs should be encapsulated together to alleviate some of the complexity of establishing the new protection environment. One can, for example, group "system routines" into classes of domains which require similar access, thereby requiring only one access control term or capability for each group. With appropriate constraints, it might even be possible to reduce the overhead of procedural packages by having relatively few of these groups. This approach is exactly what is considered in the next chapter where problems and other useful properties of such a design is discussed.

Chapter Three

A New Model

Because of the efficiency problems related to maintaining independent protection domains for every procedure, the use of a simpler, more efficient means of separation was investigated. In this new approach, the user domain is viewed in a different light. Instead of trying to separate and protect every user environment program from every other, a separation by classes is considered. This new separation divides the environment by functions rather than by components and a new solution to the problems posed in the previous chapter, based on this new division, is proposed.

3.1 Functional Components in the New Model

The basic idea of this new approach is to reexamine the user process and functionally divide the programs in it. In figure 3.1 the gross functionality boundaries are depicted. In this picture it is easy to illustrate the goals of the supervisor simplification work discussed in appendix A, that strongly motivated this research. This goal is simply to move the user/supervisor boundary down to the resource sharing and multiplexing functions. By examining the picture, the effect that moving this boundary has on the size and complexity of the user domain is clearly demonstrated. Moving the boundary places more functionality

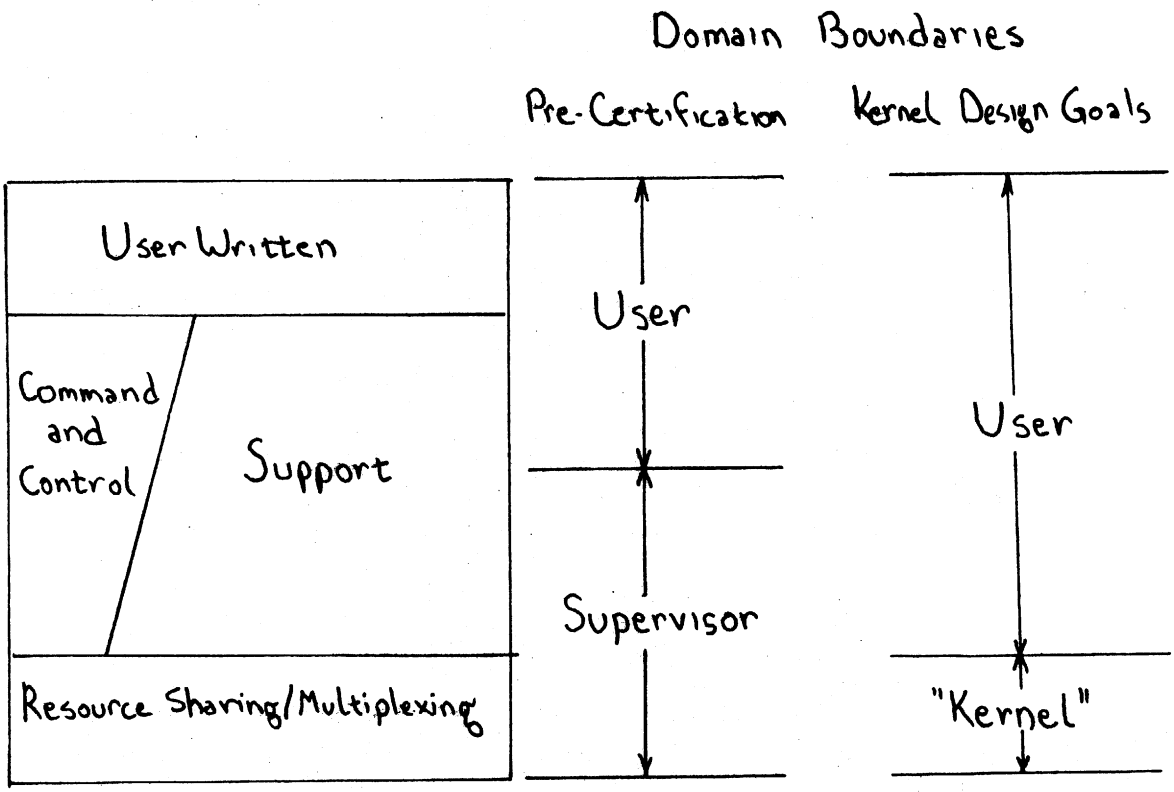


Figure 3.1 Functional Boundaries

in the user domain, thus increasing the harm that an unchecked program error can cause. For the rest of this work it is assumed that the user/supervisor boundary is in fact moved to the point established as a goal for the supervisor simplification effort.

The user domain is considered to be broken down into three classes of programs: command and control, support routines, and user programs. The command and control programs are those programs that allow users to control the execution of their processes (e.g. stop and start) and to tell their processes what to do (i.e. what programs to run). These programs are unlike any other programs in the user domain and are

therefore considered independently in a later chapter. Until then, the command and control programs shall be ignored. Procedures that fall into the support routine class are those that serve no other purpose but to provide a more elegant abstract machine for the user programmer. These procedures include stack management, call - save - return macros, Input/Output access methods, software for floating point support, conversion routines, free storage management, dynamic linking, and namespace management, to name a few. These procedures are usually provided by the system itself. The user programs are those programs written by the user to perform a desired task, and make extensive use of the support routines.

Partitioning the user domain in this way allows the support routines to be protected, as a group, from the user programs. It is assumed that the support routines are tested and debugged, and seldom change. Thus, a manual review of the support routines can verify that they will not interfere with each other. Therefore, protecting the support routines as a group should pose no additional problems.

The support routines are common routines, shared by the user programs. A failing support routine would have a more widespread effect than a single failing user program. Thus, protecting the support routines is in fact protecting some of the common mechanisms of the user domain.

Protection of the actual code is not the only thing that is wanted though. Protecting those components of the user execution environment discussed earlier so that the support routines can function independently from the user programs is desired. The support routines cannot be allowed to share those components with user programs. However, because the correctness of the support routines can be assumed, the support routines can be allowed to share the environment components among themselves. In this way, breakage and therefore inefficiency in memory utilization is held to a minimum above that of the original combination of user and support routines. This compares favorably with the ideal solution proposed in the previous chapter, where no such sharing was possible. The "working set" [Denning 68], or number of pages of memory required by a computation in this scheme would be a maximum of twice the number required by the original, single environment implementation. This is so because there are only two domains where the pages can be and in each domain those pages can be as packed as they were in the original implementation.

In summary, two environments have been created within the user domain. The next step is to choose a protection mechanism for keeping the environments separate. By examining the dependencies and interactions between the two environments, a suitable mechanism can be decided upon.

3.2 Deciding on a Protection Mechanism

The types of functions performed by the support routines are simple and likely to be invoked often. With functions such as the call-return sequence and stack management implemented as support routines, it would clearly be impossible to have them exist in an independent address space where some outside active agent performs the communication between the two, since these functions are necessary to perform the communication. Thus, a "distributed processing" approach involving parallel processes is unfeasible.

Some form of linked address space between the two environments is needed so memory can be shared. The support routines must be able to manipulate elements of the user programs (stack, linkage, etc.). On the other hand, it is desired to prevent programs from interfering with any of the elements of the support environment. This type of nesting of privileges very closely parallels the user/supervisor modes discussed earlier.

At first, placing the support routines in the supervisor might be considered. However, appendix A presents reasons why programs which are not necessary for correct operation of the system should not be in the supervisor. Recent research in system certification [Schroeder 75] has crystallized these reasons. Appendix A contains a brief description of this work.

However, besides certification there are very simple reasons for not placing programs in the supervisor. These reasons have to do with extendability and maintainability. Sophisticated users of Multics greatly appreciate the ease with which they can change and replace almost all parts of their execution environment. If the code were in the supervisor a user dissastisfied with the particular implementation would be unable to change it. Furthermore, bugs in the code would require supervisor changes to correct, which is more often a harder job than replacing user programs. A simpler supervisor is obviously more easily maintained and upgraded, and can be more quickly learned by new system support personnel.

This point has been reached by noting that the nested privileges offered by the familiar user/supervisor modes would be useful for protecting the support routines. But reasons for not actually making use of this scheme have been pointed out. However, the concept of "rings of protection" [Graham 68, Schroeder and Saltzer 72] can be used here with much success.

Rings of protection, or more simply just rings, are a generalization of the supervisor/user domains discussed earlier. Rings are an ordered set of protection environments such that ring j has at least as much access to data and programs in rings j + 1 through the maximum ring number as each of the higher rings themselves do, but only controlled access (call, read, or no access) to data and programs in rings 0 through j - 1.

The domains of interest here are totally ordered in terms of access privileges. Therefore, rings can be used for their separation. Thus, for example, the supervisor can be put in ring 0, the user environment in ring 2, and have the critical programs of the user execution environment in ring 1. This proposed structure is shown in figure 3.2 below. The user support routines are protected from user programs in the same way that the supervisor is, and yet are not part of the supervisor.

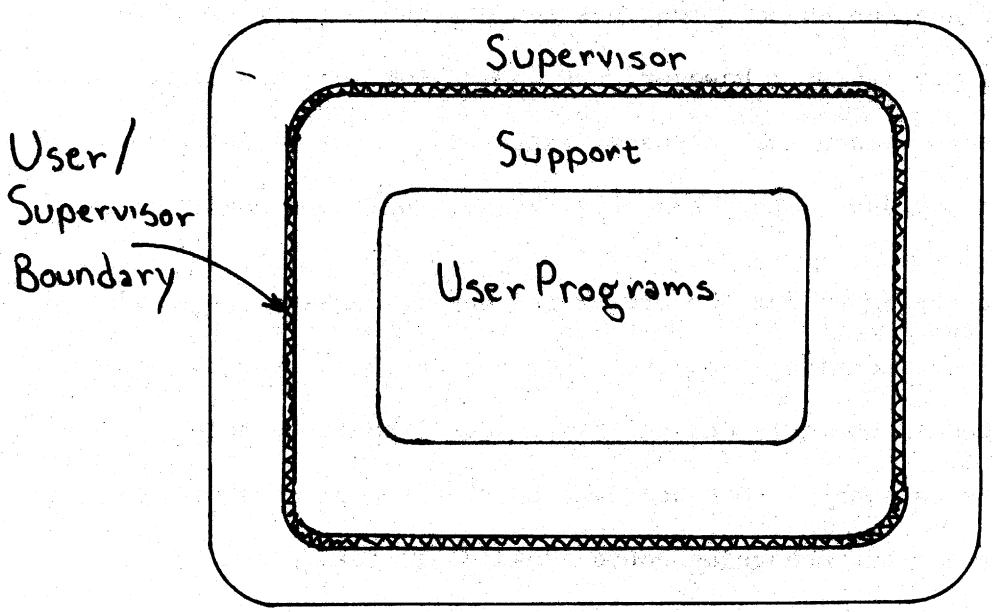


Figure 3.2 Proposed Structure

The significance of this last statement must be emphasized since it is a major design goal of this research. In a correctly designed system, the supervisor contains only those programs that enforce the security of the system. Outside the supervisor, no code can do anything

to interfere with the supervisor or another user. So even though the support routines are protected from the user written code, they are outside the supervisor and thus cannot affect system security. This is a major aid in assuring the correctness of the inter user protection mechanisms or system security.

From the user standpoint, having the support routines outside the supervisor is beneficial in that users can be granted the privilege to modify or replace the support routines so that their environments can be tailored to their needs. However, we need not be concerned about malicious users since as already explained, they cannot affect the supervisor of any user; they can only degrade their own environment.

Now that the model and the protection mechanism have been chosen, it is time to consider how the finalized design solves the original problems stated in chapters one and two. Recall that the major goal was to trap errors as soon as they occurred so that the process would not be destroyed and so that debugging could immediately take place.

With the proposed design, any attempt to wildy store data on the support routines themselves or in their environmental components, would cause the hardware to "trap" the offending instruction, preventing it from continuing. An error condition would then be signalled (see chapter six for more information on signalling), notifying the user that the error had occured. The exact instruction and location causing the problem could be determined and subsequently fixed either by patching and continuing, or rewriting and recompiling the program.

Although it may not always be possible to continue, at least the process as a whole would be saved. Saving the process allows the user to continue working on the development of the program and prevents any damage to databases that might have occured if the error was not caught.

The ideal solution would work just as well but would extend the protection to each and every program. However, as stated earlier, the support routines change infrequently and could therefore be checked to insure that they operate properly together. Thus, the only protection needed, at a gross level, is for the support routines, allowing the robust environment for program development to be realized.

Chapter Four

A Closer Look at the Support Routines

This chapter presents more detail on identifying and protecting support routines. It describes two methods of protection: One for procedures that have no static storage, and another for the more complex functions of the user environment with static storage.

4.1 Two Cases of Support Routines

Some of the low level support routines on a system require no static storage, linkage, or databases for their operation. Simple mathematical algorithms are examples of such routines, where, for instance, only processor registers are used. The only protection necessary for these type of routines is for the code itself. Systems that feature direct sharing usually provide low level support routines in a shared area that is not writeable by any process. This obviously prevents one user from interfering with another. This same protection also prevents users from writing over the code and harming themselves. Therefore, rings are not needed for these routines; but on systems where every user gets a personal, writeable copy of the routines, rings can be used.

The more complex and interesting protection is for those routines with static storage (either internal static, linkage, or external data bases). (8) In these cases, those static elements must be placed in a lower ring for protection. In order for the routines to access the areas and function properly, they too must be in the same ring as the static areas (or lower but would have no additional advantage from being there). Code in the lower ring does not prevent the sharing of that code between users, since the code is (or can be) pure; The impure sections (linkage and internal static) are allocated per process, however.

4.2 Guidelines for Support Routine Coding

Dependencies among routines must be established to insure that no procedure in the lower ring depends on (uses) a procedure in the higher ring for its correct operation. If this occurs, then the module in the higher ring must also be brought into the lower ring.

Once a module is brought into the lower ring, suitable entry points are made into gates, allowing the outer ring programs to call the inner ring procedures only at the specified entry points. Internal interfaces are thus protected.

---

(8) If allocate/free areas are used some static storage is required to remember the location of the allocated block. Otherwise, if the blocks are used only during the execution of the procedure and then discarded, it is essentially being treated as automatic storage.

Parameters that are passed on user calls to gates are validated by the ring mechanism to insure that access to the parameters is allowed in the calling ring. This prevents the calling ring from specifying areas that are inaccessible in the outer ring, but accessible in the inner ring. Without such checks, the outer ring could declare that an area used for static storage in the inner ring was the area for an output argument, thus causing the inner ring to destroy itself. This problem is exactly the same as that faced by a user/supervisor interface [Schroeder and Saltzer 72] except that here the protection is mainly for self protection and debuggability, not system integrity.

Modularity of the procedures [Parnas 72a, Liskov 72] is necessary to help in identifying and separating the user domain into functions so that appropriate ones may be protected. Modules that interact strictly by standard parameter passing via calls to procedure entrypoints are likely to be the best candidates for separation and protection. However, modules that interact via a shared database pose a problem. Although parameters are usually passed internally as a shared database, there are specific rules for dealing with those parameters. Module specifications can specify the range of, and the legal manipulations to be performed on the parameters. However, there are no specification techniques yet devised to specify a limited set of operations when dealing with shared databases directly. Thus, the boundaries of modules that share a database are not well defined and the dependency between modules is hard to establish. [Janson 76] discusses this in some detail and has termed modules that share a database as <u>weakly modular</u>.

The way that weak modularity prevents protection of a module is straightforward. If one module is placed in an inner ring, it still depends on the shared area to operate correctly. The shared area must remain in the outer ring so that the other module can manipulate it. But if the area is in the outer ring, any program there can write on it and therefore affect the correct operation of the inner ring module.

Support routines that can be protected are therefore limited to those that do not interact in ways other than through the standard call/return mechanism. A poorly designed system (i.e. one in which shared databases are the usual means of communicating parameters) can thus limit the number of modules protected. Interestingly enough, the goals of "clean modular programming" exactly identify those modules that can be protected. Functional abstractions and data hiding both provide for the type of modules that are acceptable [Parnas 72a, Liskov 72].

4.3 Examples

This section presents examples of support routines that may be in the supervisor but could be moved to the user environment. Although the routines could be in the supervisor, they are not there due to a desire for a well designed, certifiable system. Examples chosen are event management, timer management, Input Output management, and namespace management. This section also describes how rings can be used for the direct protection of linkage.

In this section, it must be rembered that should one of the support routines fail, the process would no longer be able to continue program development and debugging. The reason why keeping the current process is considered is due to both the time and expense that went into creating the process, which may contain a considerable amount of volatile state information useful for continued work. Therefore throwing away the "broken" process and acquiring a new, good one is less desirable than continuing with the old process. The failure modes considered in this section are those due to "wild" storage of data in sensitive databases needed for continued execution of the process.

Interprocess communication can be accomplished by direct writing into shared memory. However, to avoid busy waiting (9) processes can go to sleep or "blocked" and wait to be "awakened" by another process. Only the minimal amount of software necessary for the actual process blocking and awakening need be in the supervisor. Multiplexing the blocking for various events can be handled by the user.

In use, a process goes blocked and is resumed when <u>any</u> event is sent to it. The user block routine then requests from the supervisor a list of all the event messages that were sent. If the one that the process was waiting for is in the set, the user block routine returns to the waiting program. Otherwise, the received messages must be saved for future reference, and the process goes blocked again.

---

(9) Looping while waiting for an event to occur.

The integrity of the area containing the list of events known by the process and those messages that have arrived but not yet processed is vital to the normal execution of the process. Thus this area and its manager are ideal candidates for protection.

Another multiplexed mechanism can be real and virtual timers. Suppose that the supervisor provides for only two timers per process, one for absolute time and the other for elapsed virtual (chargeable) cpu time. These timers can then be multiplexed in the user domain thus reducing supervisor complexity. One way of accomplishing this multiplexing is to maintain a list of timers in use, sorted by "alarm" time, with the time closest to the present in the actual timer supported by the system. When that timer goes off, or when a closer time to the present is added to the list, the real timer will be set to the real closest time to the present. This list of timers for the timer manager is considered another important, but fragile, support facility of the user environment.

If the hardware of a system is correctly designed, and users do not share I/O devices, only a minimal mechanism need be in the supervisor to support user requested input and output. Users can be allowed to write a channel program to control "their" devices and no others. The supervisor need only start the channel for the appropriate device and possibly assign storage for the duration of the operation. (10) Only a

---

(10) Multics allows users to request that the supervisor not page out a page of memory for a short duration so I/O to fixed addresses can be accomplished.

simple interface to the supervisor is needed to identify a set of I/O instructions for a particular channel to execute. Device control modules should be in the user environment along with all device access programs. Higher level database access programs and code reflecting a device independent environment should also reside in the user domain. Of course I recommend that all these functions reside in the protected portion of the user domain.

Multiplexed devices can be harder to handle. If two users are allowed to share portions of a single device (e.g. sections of a single disk pack), it would be impossible to keep this code in the user domain. By definition it must reside in the supervisor since it deals with resource sharing and multiplexing among independent users. However, in the case of one controller with multiple devices (drives), correct design of the controller can allow it to be shared by many users. All that is required is to prevent an I/O channel program from switching devices during its execution. Then only allow the supervisor, at the time the channel program is started, to specify the device on the controller. This approach has in fact been used by Honeywell on their single controller, multiple drive tape units for Multics [Greenberg 76].

Short reference names are local user defined names that can be associated with long global names to simplify talking about an object. Once the relationship between a reference name and global name is made, only the short name need be used. As an example, a reference name might be "square_root" for a procedure that computes square roots, while the

global name might indicate the location of that procedure within a naming hierarchy such as:

"ROOT>system_library>math_routines>square_root". (11)

Typically a linker, or binder, associates the reference name as used in programs (e.g. x = square_root (y)) with a particular module in the system. As Bratt [Bratt 75] explains in his thesis, the reference name facility need not be in the supervisor. Bratt describes how that facility can be removed from the supervisor and be placed in the user domain. However, this facility is greatly depended on by all programs in the user domain. If the reference name manager should fail, inter-program linkage would fail and no new programs could be found or executed. Thus it is imperative that this facility be protected from damage by user programs. Placing the reference name manager in the protected environment is essential to the goal of providing an "unbreakable" user environment.

All the above modules contain significant state information in static storage. Therefore, it would be desirable to place those programs in the protected environment so their important state information is protected.

Finally, consider dynamic linking. Janson [Janson 74] explains in detail how dynamic linking can be removed from the supervisor of an operating system. However, here too, the eventual placement of the

---

(11) A>B indicates B is in directory A. ROOT is the root directory of the naming heirarchy.

linker module is in the user domain. The set of search rules, guiding the linker to find a specific copy of a named procedure, constitutes a static database used by the linker. To protect the search rules, and thus the linking mechanism, the linker should be placed in the protected environment.

Another part of the linking mechanism that Janson termed environment initialization is also important to protect. In Janson's design, a procedure locates its static storage and linkage section when it is entered. The first time that the procedure attempts to do this will trigger a mechanism which will allocate and initialize these sections. This allocation and initialization processes can be protected.

The tables that procedures use to find their linkage and internal static sections, the Linkage Offset Table and Internal Static Offset Table (LOT and ISOT), can also be protected. The LOT and ISOT only have to be read by procedures; it is an error if a user program writes in them. Similarly, linkage sections are only read by programs. Thus the LOT, ISOT and linkage sections can, and should be protected from errors caused by user programs. To do this requires placing not only the environment initializor in the protected environment, but the linker as well, since the linker modifies linkage sections. This approach realizes at least part of the original goal of protecting the components of the user environment.

Chapter Five

Command and Control Routines

A human user of a system is aware of other environments besides the execution environment. Any program that accepts input from the user's console interprets what the user types in a different way; hence the appearance of different environments. This chapter examines those environments that are used to control a process. This research has identified two environments used for this purpose, which have slightly different properties. These environments are embedded in the code that was classified as command and control programs in an earlier section. These two environments and their properties are discussed below.

5.1 Structure of Environments

The relationship between the user at a terminal, the program execution environment and the command and control environments is shown in figure 5.1 below. Between the terminal and the program execution environment flows user program input and program output. Between the terminal and the command and control environment flows program loading, stop, and start requests, as well as messages from the command and control environment (e.g. "program not found"). Finally, between the program execution environment and the command and control environment flows a program generated command stream and control messages such as

start, stop, and load a particular program. All these streams will be discussed in more detail in later sections.

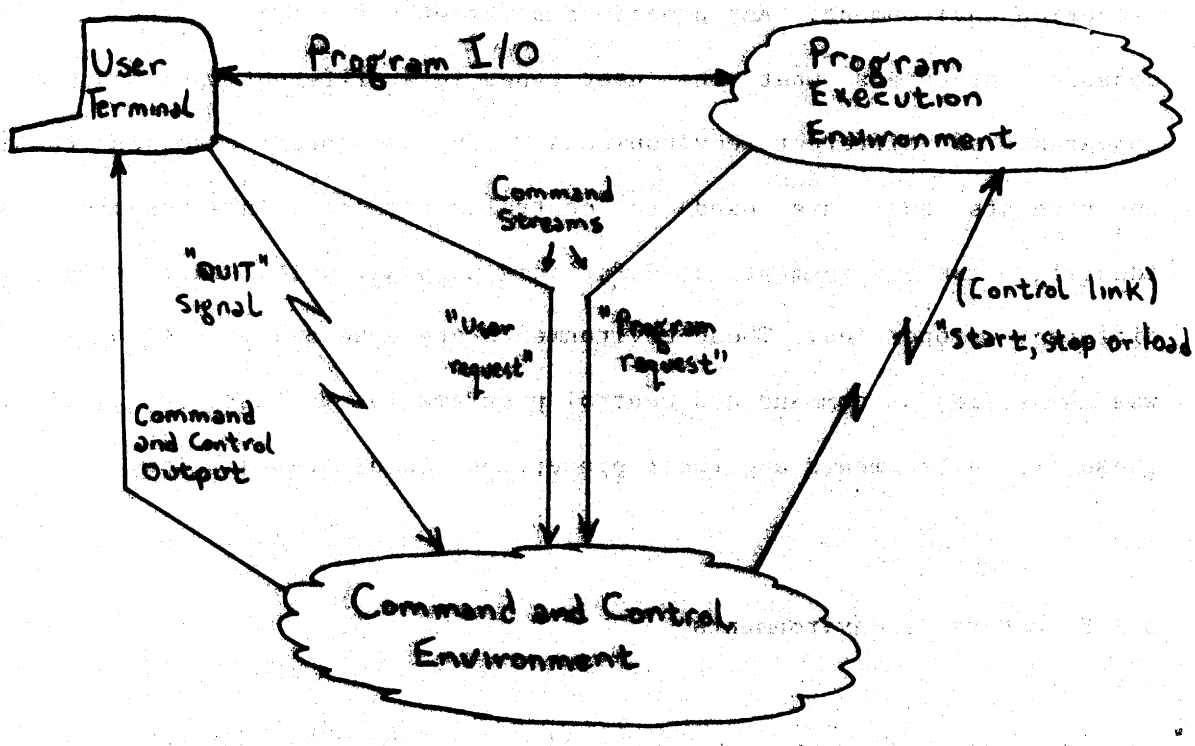


Figure 5.1 Command and Control Structure

Consider the components of figure 5.1. The program execution environment has already been discussed in previous chapters and the reader is probably familiar with some terminal on a timesharing system; thus, an examination of the command and control environment is needed to complete the picture. Although these environments are shown merged into one in the figure, they have different properties in reality. An attempt will be made to describe these differences, but as you will see, it is hard to separate them entirely. They will then be considered merged for the rest of the thesis.

5.1.1 Command Environment

The command environment is seen as the program that the user is in communication with immediately after being given a process. In particular, the command environment responds to user requests that describe what programs to run for the user. After the specified programs have completed, the user is again in communication with the command environment. Typical requests to the command environment are "run the editor" or "compile my program" in whatever syntax is understood by the command environment.

The types of messages that are sent between the command environment and the rest of the user domain are basically "load this program and transfer to it" in response to a user command, and in the other direction "execute this command line as though the user typed it" for programs which generate command lines. These messages are shown in the figure below as user requests and program requests, respectively.

User requests are generally familiar to all computer users. It may be some form of job control language ("// exec pgm=basic") or one word command lines on a timesharing system ("basic"). The use or need of program requests might be doubtful though. However, use of this feature could be made, for example, for the implementation of a command file facility (more will be said about this in a later section). This

facility allows users to create a file containing a sequence of commands to be executed, and invoked by running the command file program specifying that file as it's input. The command file program then simply calls the command and control environment with each command line in sequence. The command lines are passed to the command environment via the program request stream.

Another use for this stream is to allow programs to pass a "canned", or user supplied command line to the command environment under special circumstances. This feature might be employed in a procedure that could accept a generic command string and append or substitute generated information into the string. Then the command processor would be called with each such generated command string thus relieving the user from having to type the same command over many times. An example of such a procedure would be one that simply generates a list of new or modified information files; invoking this command would generate a list of modified or new file names. If the user wanted each one of the new or modified files to be printed, he/she could supply the generic command string "print %" to the list_new_files command, which would then generate the commands "print new_file_a", "print help_file_3", ... (etc.). The "%" in the generic command string would be replaced by actual file names in each generated command. Then the command processor would be called with each of the formatted command lines. (12)

---

(12) This is just an example and not a proposal.

5.1.2 Control Environment

Whereas the command environment quietly waits for requested programs to complete, the control environment is always awake and listening to the user (at least conceptually). The exact mechanism that is used is not important here; what is important is that at any time the user can say "hey you (computer or process), stop and talk to me !". In this way, the user can stop an infinite loop in a program and not waste time and/or resources waiting for a failing program to complete. Along with the power to say "stop", the user would also like to say "OK, go ahead", or "forget that".

Functionally, the user generates a signal causing the process to enter the control environment. (13) Users then have a choice of debugging and continuing, or forgetting the computation. They may also run any other program first, before returning to the stopped program. In this way, a calculator program can be used to check intermediate results. Similarly, an inter-user message facility can be used to ask the author of a program "hey, what's wrong ?", get a response, then continue the interrupted program. This general program calling can be implemented by allowing the control environment to call the command

---

(13) On Multics, users generate this signal by using the "break" or "attention" key on their terminals. On TENEX, control-C generally does the same thing.

environment to process all but control environment requests. However, control environment requests might actually be parsed by the command environment and invoke control environment functions. (14) This apparent double dependency can be resolved by joining the two environments, thus providing a single command environment to the user which is simpler in structure and more easily understood.

In summary, the major difference between the control and command environments is that the control environment is asynchronous with the rest of the user process flow, and responds to simple, base level requests, with possible extensions to include full command processing. For the rest of this thesis, the two environments are considered merged into one.

5.2 Processing Commands

This section examines the details of how a command may be processed. There are, of course, numerous ways to accomplish the type of processing described in this section (see [Broughton] for example), but the ones chosen are useful for explaining some protection problems that will be described later in this chapter. The processing described covers a large variety of known systems.

---

(14) This is done on Multics where the command environment processes all requests, some of which invoke control environment modules.

In an overall view, the command environment performs four operations in response to a user request to execute a module. First it parses the command line and performs possible substitutions, parameter evaluations, and conditional evaluations. It then searches for the identified (possibly ambiguously named) module using a set of search rules. The module is then brought into the address space of the user. Finally, the module is transferred to and it begins executing.

Note that none of these functions require that the command environment be part of the supervisor; no special privileges are needed to perform all of the functions stated. Even so, some systems [TENEX, CTSS] have the command environment as part of the supervisor. Placing the command environment in the supervisor suffers from the problems discussed in chapter three. If the reasons for having the command environment protected is solely for the benefit of the users (i.e. they cannot destroy the command environment), then the discussion on protection, later on in this chapter, should help in choosing a new approach to solving that problem.

The functions of the command environment are now examined in detail. The tranlating and parameter evaluation mechanisms are examined first because they are particularly important in this work. A modular design of command processing is described. The modules are classified into two categories: 1) those that look at every command line, and 2) those that are optional and look at command lines only when asked. This classification will be referred to later on in this chapter when the protection of the command and control environment is discussed.

5.2.1 Text Substitution

Translation, or substitution of text, can be a complicated task if it is not restricted. What is meant here is simply the substitution of one string of characters for another in the command line. A frequently used form of this type of processing is for abbreviations. For example, a user types "fortlm" and gets "fortran -list -map" which might execute the Fortran compiler and provide a listing and statement map. This type of processing can be neatly packaged in a "front-end filter" preceding the actual command processor as shown in figure 5.2 below.
This approach is useful because the abbreviation processor can be coded in a separate module (allowing easier debugging) and inserted only if desired. This module is considered a member of the "always used" category since, once it is selected and inserted in the command processing path, it examines every command line to determine if substitutions are necessary.

5.2.2 Parameter Evaluation

A second aspect of command processing that is considered here deals with how commands can be affected or controlled by the user's total environment. With this flexibility users can control the execution of a command based on the date, a list of files in a catalogue or directory,

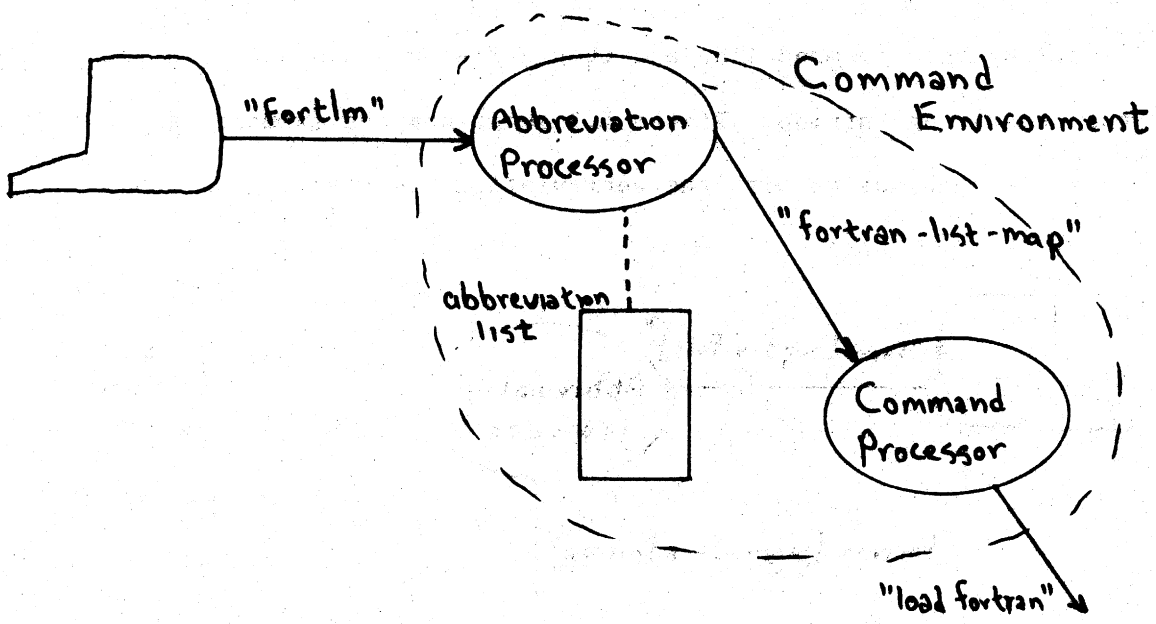


Figure 5.2 Abbreviation Processing

or on the name of the person who sent them the last message (for example). Use of these controls easily allows printing of only new messages (those that have been created today), compilation of all FORTRAN programs in a certain directory, or replying to the last person who sent you an interactive message, without ever having to specify the date, list of files, or the name of the last message sender.

Such features can be implemented as a special cases in the command processor and use special keywords. However, the most general approach is simply to call procedures which implement each function and return a string to the command processor as a result. Such an approach is used

on Multics; these special functions are referred to as "active functions" [MPM]. The active function processor evaluates all active functions in a command line and then calls the command processor with the resulting string. Figure 5.3 shows how the example stated earlier works in combination with the abbreviation processor.

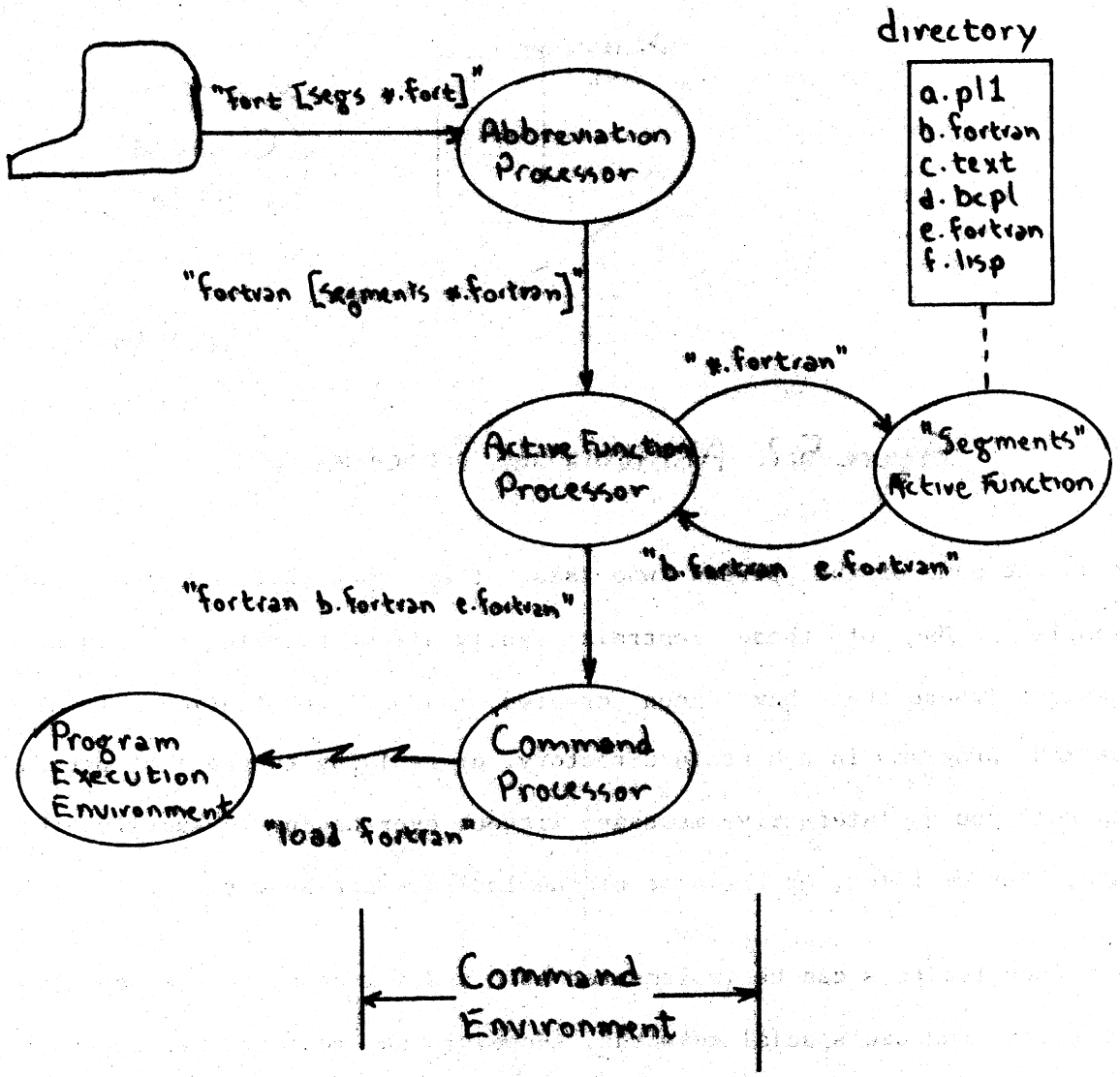


Figure 5.3 Total Command Processing

The program invocation mechanism of the active function processor and command processor are very similar since both call procedures. Therefore it is possible to merge these two functions into one. (15) However, the modules will be considered independent to facilitate the protection discussion later in this chapter. The active function processor also looks at every command line to see if active functions must be invoked. Thus, it too belongs to the "always used" category of command processing functions. Notice that the active functions themselves are not always used and thus the collection of active functions are members of the "optional" category.

5.2.3 Command Files

On many systems long sequences of commands may be stored in a file and executed by only typing a single command line. This type of processing is useful for reducing typing time and eliminating errors, for providing complex "abbreviations", and for providing a simpler interface to a complex system (e.g. catalogued procedures). Most such implementations allow parameter substitutions and offer a language to control execution and flow within the command file (e.g. "if ..." and "go to ..."), including error handling (e.g. "on error go to ..").

---

(15) This is done on Multics.

Such processing can be merged into the command processor, but for the sake of modularity and clean design it should be a separate module which reads the command file and calls the command processor with each command line. This type of processing also need not be protected in any way (at least for system security reasons) and therefore can execute in the user domain making use of the "program request" stream described earlier. If implemented as a separate program, the command file interpreter does not look at every command line; it is only invoked when the user specifies that it should be. Thus, the command file interpreter is considered a member of the "optional" command processing functions.

5.2.4 The Command Processor

After all the command line processing is complete, the command processor is called to invoke the specified program. The program name in the command line is used to find the actual procedure within the naming heirarchy. The linker search mechanism may be used for this purpose which, if used, would provide a single search strategy in the user domain for both dynamic linking and command program locating; one search strategy is obviously more easily remembered than two.

The parameters supplied on the command line are formatted appropriately for passage to the specified procedure. The "load program" signal is sent to the program execution environment, which may

cause the program module to be read into main memory, or merely assign the module a virtual address (making it "known"; see [Bratt 75] for details). Finally, the "go" signal (which may simply be a transfer instruction), starts the loaded procedure.

Obviously, the command processor is the key module in the command environment and thus is a necessary and always used function.

5.3 Getting to the Control Environment

The simplistic approach to entering the control environment is merely to transfer control from the executing program directly to the control environment procedures, much like an interrupt sequence. However, there are times when users wish to program their subsystems with internal "attention" procedures which get invoked at the time a stop request is issued from the control environment. These procedures can be used to cancel the effects of a request that is currently being worked on, or to make a database consistent (e.g. unlock it). For example, when LISP on Multics [Reed 76a] recognizes that the user wishes to stop the computation, it first updates all bindings in memory from the working registers so that the most recent effects will be seen by the user without explicit knowledge of register optimization built into the LISP subsystem. Only then does LISP allow the "stop" to take effect.

However, if the subsystem is malfunctioning, it would be impossible for a user to abort it and return to the command environment. Therefore, there is a need for at least two kinds of stop signals; one which allows attention procedures to be invoked, and a second which goes directly to the control environment. Then users can first attempt the more elegant stop, allowing the procedure to recover, but if that fails, they can use the "panic stop".

To implement this type of feature there must be a way of specifying the type of stop desired. The "break" or "attention" key found on most terminals is usually used as the "stop button" but there is only one of them. This can be multiplexed by having users type a single character after the break key denoting what type of stop is desired, or by some coding in the number of breaks sent. The latter approach is prone to misuse however, because impatient users would wonder whether a single break got through if no result of that fact is quickly demonstrated and then would send a second break which would get them into the wrong environment. The TENEX system [TENEX] has the useful feature of allowing all control characters (16) to generate a variety of process interrupts. In this way no multiplexing of the break key need be done, and simple one character strokes can effect desired responses. Each control character can then invoke a special independent function. TENEX, for example, responds to control T by first beeping (indicating that the system is still there), and then giving a system load estimate and resources used since the last request.

---

(16) A keyboard character sent with the control (CTRL) key pressed simultaneously.

5.4 Implementation of Command and Control Environments

How the command and control environments are implemented is of concern because they may be implemented as user programs running unprotected in the user execution environment, as on Multics. If this is the case, then these programs are also subject to interference from other user programs as discussed earlier. Control of a process is an essential feature; therefore in providing a robust environment it is necessary to protect the command and control environment. How these programs can be protected is now considered.

5.5 Protecting the Command and Control Environments

Protecting the command and control environment means that the procedural packages implementing that environment must be protected. The choices for doing this basically fall into two categories. The programs can exist in independent processes with separate address spaces, or they can share an address space and memory with the user programs, as the support routines were allowed to do. The types of interactions between these programs and the others of the user domain can help in deciding which one to choose.

The messages to the command environment are simple character strings. For the control environment, messages are, more or less, "stop" and "go". Because of the nature of these messages, they are sent infrequently (usually once per user request). For this reason tight coupling, by memory sharing, between the two environments is not necessary, and efficiency of communication is not that important. Thus, virtual processors, or physical separation might prove to be feasible, particularly in light of the current trend towards "distributed computing".

The National Software Works project [NSW] uses the approach of a "front end" computer as a user to computer network interface. The front end has some memory and a moderate amount of processing power. It can parse user requests and format them into a more rigid syntax easily processed on a variety of host computers. Thus, front end processing can alleviate some load on the host. The front end can also provide a more reliable computing utility by having low level software choose different computers to perform actual requests, in the event of failure. The Distributed Computing System [DCS] has this goal in mind.

A front end processor can also support local editing, allowing a user to compose and edit text without making use of the host computer, thus further reducing the load on the host. Character at a time echoing can be supported by the front end with special "action" keys forcing the transmission of words or lines to the host. This approach alleviates the need for the host computer to respond to each character typed by

each user and thus also helps in reducing system load. This last facility is in fact currently being implemented on the ARPANET [ARPANET, RCTE].

There are, however, problems with this approach, one of which has to do with the program "loading" feature of the command environment. The other problem is that some form of communication is needed between the control, command and user environments. The loading function and communication mechanism can, of course, be in the supervisor, but arguments have already been presented for not placing similar basic functions not relating to system security there. The best place for these features is in the user environment, but as mentioned many times, programs are subject to failure there. To prevent this, the communication mechanism at the user environment end, and the program loading function, can be placed in the protected support routine environment described earlier.

Another reason for not using multiple processes is that the "stop and go" features of the control environment can very simply be implemented if there is only one execution point in the user's computation. It is obvious that if some signal from the user causes immediate transfer to the control environment, the user program will automatically be stopped (much like an interrupt). Similarly, when the control environment transfers back to the user program (much like a return from interrupt sequence), the "go" function is obtained.

There are arguments for not having interrupts, however, mainly based on complexity and design of programs that service them. The environment that the interrupt handlers run in is generally fragile and not completely specified due to possible functions affecting it that were in progress at the time of the interruption. The preferred option is to use processes where appropriate. These processes simply wait for the desired signal, then act accordingly in a synchronous manner. When the job is done, the process then goes "blocked" waiting for the signal to occur again. In this way, the environment that the interrupt handlers run in is well defined.

This scheme does not eliminate the need for interrupts, however, but limits the code that must be run during the actual interrupt processing to that which performs scheduling functions. By nature, these scheduling functions are designed to execute in a manner that does not require full system capabilties. For the actual stopping of the user execution environment, real interrupts must also be used so that control can be torn away from the executing code.

I have no argument against the use of processes for such functions except that on a large system, like Multics, where processes are very powerful processing agents, the expense is simply too great. It would be advantageous to provide cheap, weaker processes to perform these functions for simplicity of coding and understandibility. How processes can be implemented cheaply is discussed by Reed in his thesis [Reed 76b]. Such processes were used by a memory management design with

much success [Huber]. Lacking these cheap processes, it was decided to place these functions in the environment already set up for the support routines. This gives us the simple control over the process discussed above (because of the single execution point), and requires no additional tools for multi-process intercommunication to be developed.

5.6 Design Decisions Based on the Protection Scheme

The following sections describe decisions that were made solely because of the decision to place the command and control environment in the protected half of the user domain. In using a different approach, such as multiple processes, it is not immediately clear that the same decisions would have been made although given some thought, it would seem that they are not totally unreasonable because of other criteria such as delay time and load transfer to the front end processor.

5.6.1 Command Processing Revisited

As a result of placing the command and control environment in the protected half of the user domain, design decisions have to be made regarding the placement of each of the command environment modules. The major factor that influenced the decisions discussed below was the desire to provide a path to the command processor that was unaffected by failures in the unprotected half of the user domain so that control

over the process could be maintained. Thus the "always used" modules, categorized earlier, are precisely the ones that must be in the protected environment. The "optional" ones need not be, however. Unlike the decisions made in the certification work discussed in appendix A, there is no exact minimal set of programs that have to be protected. The choice can therefore be based on considerations other than security. One rule that can be used is "if it can be protected it should be", but this might lead to protecting the entire collection of programs in the user domain. While this is not a real problem, it clutters up the protected environment with many simple programs that are not essential support modules. Although it would be desirable to protect all programs, as in the ideal solution presented in chapter two, we must remember that the preservation of the process and control over it is of paramount importance and the loss of a simple function could be remedied dynamically when it is discovered. Keeping the protected environment simple also helps in understanding and maintaining it leading to a more robust environment.

Obviously the command processor itself must be placed in the protected environment, as it is the essential component of the command environment. The active function processor must also exist in the protected environment because if it fails, no command line will get through to the command processor. However, the active functions themselves should not execute within the protected environment for three reasons. First of all, they may be user supplied and might be in the process of being debugged. Secondly, the protected environment is

different from the one the user "sees". The working environment is supposed to be the one that controls the execution of the command. But by executing in the protected environment, active functions refer to the wrong working environment. Within the protected environment the active functions have access to more areas than the normal user programs, and know about more files or information than the user intended. Consequently, they might specify operations that should not or cannot be done (e.g. compile the command processor because its name matched "all PL/I programs"). There can also be naming conflicts between the protected environment and the normal execution environment. The user may request the compilation of a new version of a program but get the old one because it is found in the protected environment. Finally, active functions are not required in order to pass a simple command line to the command processor, which is all that is desired for process control.

The abbreviation processor must exist in the protected environment because it looks at every command line and could prevent any commands from getting through to the command processor if it fails.

Finally, the command file processor need not be in the protected environment since it is not used for simple command lines and can therefore execute in the unprotected part of the user domain without fear of loss of control over the process. The choice of placement of this module was just a matter of taste and could have, just as easily, been placed in the protected environment. However, the decision for its

placement was also due to a desire to not clutter up the protected environment with non-essential support modules.

5.6.2 Command Processor Escape Mechanism

Since the placement of the command and control environment programs are in the protected environment, some user commands have to be executed in the protected environment. In particular, all control environment commands such as "start", to continue a stopped computation, and "release", to abandon a computation, have to be executed within the control environment. Since only one command interpreter exists, it has to know that it should treat these commands differently. Thus, a table of commands to execute in the protected environment can be used which is looked at by the command interpreter.

It is also useful to provide an additional mechanism for users to specify execution of a command in the protected environment for just that one instance. An example of such a program might be the access control setting program. Executing in the user program environment, the access control program could not affect programs in the protected environment; this could only be done from within the protected environment (this is exactly what is wanted, normally). However, a user might want to invoke the access control program once for specifically setting the access control list on a protected program, possibly for installing a new version of the command processor, and at another time

escape mechanism is necessary. Justification for allowing this and still claiming to offer some protection for the support routines is found in the belief that the type of error described is a rare one and not expected to occur. After all, it is assumed that the user programs are not malicious in nature. Thus maintaining this feature provides a single user interface to the command processor.

An important question that might be puzzling the reader at this point is "why should anyone trust user programs to behave respectably ?". This question deserves a good answer and is an important part of the overall design. Recall that all system security related programs exist in the supervisor and are protected from all users; once beyond the supervisor boundary, one user cannot affect another. Partitioning the user domain only helped the user from self harm; no additional side effects could occur. Thus, a truly malicious user, using the proposed system, could only affect the user portion of the process. No other user process, nor system program could be damaged. The proposed design is merely an optional aid to a willing user and not a must.

to merely add someone's identifier to a user program's access control list. The access control program should not always execute in the protected environment or it will have more access than it needs most of the time. This extra "freedom" allows mistakes to have more serious effects than they would have had otherwise. (17) Thus, the escape mechanism allows users to explicitly specify the times when commands should execute with increased privilege. All other times, commands execute in the less privileged user execution environment (except for those listed in the "special" list). Letting a program execute with only enough access and privilege to do its job has been a design goal known for many years and is discussed in [Saltzer and Schroeder 75].

The command processor escape mechanism coupled with the feature of allowing user programs to call the command processor seems to point out a gaping hole in the protection of the support routines and the command and control environment programs. Apparently, any user program could call the command processor with the escape mechanism and cause a failing user program to execute in the protected environment and destroy it.

One simple and obvious solution to this problem is to have the command processor recognize from which environment it is being invoked and ignore the escape mechanism in calls from user programs. However, this presents a different user interface and might possibly confuse users who wish to make use of the escape mechanism from within another program, such as the editor. For this reason, I feel that allowing the

---

(17) This is similar to the major problem reported in this thesis.

Chapter Six

Signalling

Signalling is discussed in this separate chapter mainly because it is a subject that can be factored out for simplicity. The reason for discussing signalling at all is due to the problems that appear when two domains interact so that signals can pass between them. This structure is relevant because of the method chosen to protect support routines of the execution environment.

To treat this subject properly a model of signalling is described. Extensions to this basic model are then introduced so that it is useable in real world situations. Problems with signalling are then discussed to point out basic pitfalls in the original model of signalling. A solution to those problems is suggested that is based on a newly designed language (CLU). Finally, problems explicitly related to signalling in multiple rings are presented and discussed in the context of the model, its problems, and the solution presented.

6.1 Purpose of Signalling

Signalling allows the establishment of a procedure that knows how to deal with a particular situation, usually an error condition, and invokes that procedure at the time the condition is detected. Return codes (an output parameter whose value indicates success or failure of an operation) are often used to denote error situations. Signalling differs from simple return codes because the procedure to handle the error is called at the time the error is detected and may allow the computation to proceed instead of simply undoing the computation and returning.

6.2 A Model for Signalling

The PL/I language has a facility, described in the next paragraph, for dynamically setting up, calling, and removing condition handlers [Noble 69]. The method used for choosing which condition handler to invoke is straightforward and generally familiar and thus will be used as the basic model for signalling. The signalling mechanism used on Multics is based on this structure. I believe that Multics is the only system on which the entire collection of software operates under a common signalling framework. Thus it seems reasonable to use PL/I as a model.

In PL/I, handlers for various conditions are established by the execution of the "ON" statement. Multiple handlers may be set up for any condition in different procedures. The most recently established handler will be the one that is invoked upon detection of the condition. Handlers are automatically reverted when the establishing procedure returns.

This mechanism allows any procedure to handle an error locally or pass handling on to a system default handler or handler supplied by the calling procedure. Local handling is considered more appropriate by some since the local procedure is more aware of the actual situation at hand at the time of the error. Parnas [Parnas 72b] however, describes how a high level routine may, in fact, be better equipped to handle an error than a low level procedure simply because it understands the more global context and significance of the error. More will be said about this later.

6.3 Extensions to Signalling

It has been found that being able to note the occurrence of an error but not handle it explicitly is useful. Similarly, taking only partial action towards fixing the problem might be desired. Using Parnas' example, an I/O routine may discover a read error but does not explicitly handle the error since it is not aware of the use or need of the record. It may however desire to maintain local error statistics and then ask its caller whether it should retry or ignore the operation. In these types of cases Goodenough's [Goodenough 75] PASS operation might prove useful. PASS allows a handler to "...explicitly disclaim interest in (further) processing of an exception, directing that the exception be passed on to some higher handler".

Working in harmony with PASS is the useful extension of having a handler for any condition that is not explicitly named in a procedure. The condition name "any_other" [MPM] is used on Multics for this purpose. This extension is useful in light of the previous example where the tape I/O routine wanted to detect <u>all</u> errors and just make a note of them, then <u>pass</u> them on. It would be extremely awkward to list all possible conditions that could arise at any point and have the same processing for each. Furthermore, since error conditions can be user defined, it may not even be possible to identify all the errors that could arise. The any_other handler solves these problems.

A third extension to the signalling mechanism is the "cleanup" condition [MPM, Goodenough 75]. This condition is signalled in every block that is abnormally terminated (i.e. does not execute a "return" sequence). A handler that specifies the termination of a procedure because of an error automatically triggers this mechanism in the procedure(s) implementing that operation. This allows the procedure(s) to restore the original state of and/or eliminate "impossible states" [Parnas 72b] in its operation.

6.4 Signalling Problems

With the basics explained, it is now possible to discuss the problems associated with a PL/I-like signalling mechanism. The problems all arise from the ability for a handler to be set up in one procedure that can handle conditions arising in another procedure. This "feature" was introduced to overcome the "inconvenience" of specifying a single handler in separately compiled external procedures (multiple times). This "dynamic descendence" rule [Noble 69] of PL/I violates modularity and thus understandability of programs in two ways.

First a low level procedure must know how its callers will react to errors arising in the procedure so that it will know what to expect from incomplete operations within itself (e.g. overflow). Thus, it cannot be programmed without knowledge of "layers" that use it and so it is not modular. A procedure may not be expecting any particular action from

its callers; it could depend on the system default handlers. However, any procedure may handle its own errors and unintentionally also handle errors of programs that it calls simply because it has a handler for itself.

Secondly, a high level procedure that handles errors of low level procedures must know the way in which the error was caused in the low level so that it can handle it properly (e.g. overflow results in highest positive or negative value).

A final problem arises when error handlers themselves generate errors. In this situation the wrong handler may be chosen to handle the error. Consider two procedures <A> and <B> both having condition handlers for various conditions. <A> calls <B> resulting in an error in <B>. <B> does not have a handler for that error, expecting that the system default error handler will suffice, or that its caller will know what to do. Assuming <A> does have a handler for that error (call the handler <A'>), the call stack will look like the figure 6.1 below. Now if <A'> should take an error like overflow, <B>'s handler could get invoked even though the module <A> was prepared for handling <A'>'s errors! <B>'s handler could obviously choose a completely different method than <A> for handling the error and thus <A'> will not function properly.

These problems all arise from the incomplete specification of error handling within each module. I am not arguing for not allowing higher level handlers to "handle" errors of low level procedures; I agree with

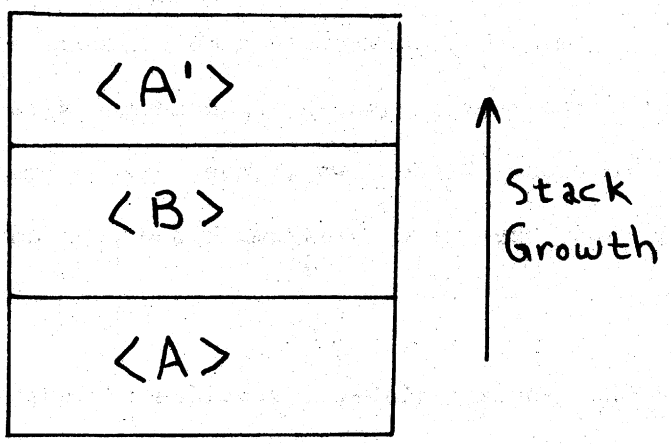


Figure 6.1 Sample Call Stack

Parnas that this is reasonable, subject to certain restrictions. I contend (like Parnas) that these mechanisms should be explicitly coded into the procedures. The "inconvenience" of doing this would be more than paid off in terms of understandability and ease of debugging of such software.

Thus, every module would explicitly detail its error handling intentions with possible options like "ON ANY_OTHER CALL SYSTEM_DEFAULT_HANDLER" or "ON ANY_OTHER ABORT" for the conditions not explicitly handled in the procedure itself. For passing of conditions upward the "pass" mechanism may be employed but I think that passing on the same condition that arose in the low level procedure is a violation of modularity since the operation of the low level procedure must be known by the high level procedure in order to effectively deal with the error. A better approach would be to have the low level call a high level routine allowing it to return values such as "abort", "continue",

"retry", or "use this answer" in a more global sense. In this way the types of responses are known in advance. More will be said on this in the next section. Either way, the error handling should still be explicit and have the programmer and program reader aware of what is going on.

The CLU language [Liskov] provides for just such explicitness in error handling [CLUnote 43, CLUnote 60]. Any error not handled by a procedure automatically causes that procedure to be terminated and results in a "failure_of_mechanism" condition to be signalled to its caller. Any upward "traps" [Parnas 72b] (higher level handling of low level errors) must be explicitly coded.

The only problem with CLU has to do with debugging. This aspect is extremely important since it is a major topic of concern in this thesis. On Multics there is a default error handler at the "base of the stack" that performs the "standard fixup" or reports an unhandled error to the user and enters a new level of command environment for debugging. Since the default error handler is called, the stack history is preserved and debugging is possible. Unfortunately, in CLU, an unhandled error terminates the procedure so no dynamic debugging is possible. CLU enthusiasts claim that "debugging mode" can be turned on thus preventing any terminations due to a failure_of_mechanism, but this has two problems associated with it.

First, debugging mode must be explicitly enabled; the Multics default handler is always there. Of course it could always be enabled but that leads to another problem. This second problem is that a normal, expected failure_of_mechanism error that can be properly handled by the calling procedure is also flagged and told to the user. Thus the computation could not continue without getting annoying messages and having to type "continue" or something semantically equivalent.

The only solution I have come up with to this problem is that termination of the procedure should not occur unless the caller is prepared to handle that condition (failure_to_[illegible]) and be willing to continue gratefully and not abnormally [illegible] also. If no procedure is prepared for this then the default error handler or user fault notifier should be called and the stack left available for debugging.

Given that such a mechanism could be put in CLU, it would seem that the ideal signalling mechanism would be available.

6.5 Multi-Ring Signalling

Assuming standard PL/I signalling is provided for in a multi-ring environment, the first problem that arises follows a simple inward call. If an error occurs in the lower ring and is not handled there explicitly, PL/I dictates that the signal should propagate to the caller. However, since the caller is in a higher ring and has less privileges, chances are that it could not deal with the actual error since it cannot affect databases in the lower ring. Thus, it seems useless to allow signals to pass outward. But not allowing signals to pass outward apparently contradicts what was said earlier regarding high level handling of errors. Since the higher ring knows more about the global situation it should have a say in what should occur if an error is detected. This apparent conflict can be resolved by the proper coding of the lower ring. The lower ring should <u>not</u> allow the higher ring to handle <u>internal</u> errors like "OVERFLOW", rather it should just indicate a logical error in the lower ring. Then the higher ring can simply indicate that the lower ring should either retry the operation or abort. It is situations like this that Parnas may have been alluding to in discussing "upward traps" [Parnas 72b], but he does not explicitly say it. Similarly, Goodenough's "PASS" operation seems to pass on the handling of the <u>original</u> error. This is where I disagree; the fact that an error occurred should be made known and a higher level should be

allowed to decided whether to continue or abort, but should not handle the original error.

It is in situations like this that Multics falls down. On Multics, if an unhandled error is detected in a lower ring the computation is aborted. The outer ring is notified of this fact but cannot ask for a retry or continuation from that point. There is currently no general mechanism for inner ring programs to specify a "checkpoint" and wait for outer ring intevention to continue. (18) The checkpoint feature allows some inner ring history to be preserved so that if continuation is desired, the inner ring need not recompute everything up to the point of the error; it would merely continue from the checkpoint. The checkpoint feature, however, should not allow the partial results to be seen by the outer ring. Any other request given in between the time of the error and the continuation or abort should function normally and independently of the partial results held in the inner ring. The checkpoint is merely a technique to help improve efficiency in cases where the precomputation involves a sinificant amount of resource usage. However, the checkpoint feature is not required for the proper operation of the signalling mechanism.

The second problem in multi-ring signalling has to do with outward calls. Outward calls is how the command environment that exists in an inner ring would call a user procedure. If the default error handler

---

(18) Errors during dynamic linking are an exception to this; they are handled as a special case and are "restartable".

should be protected as a common mechanism, signals have to travel from the outer ring to the inner ring to activate the default error handler. But there is no way to guarantee that the mechanism used to transmit the condition from the outer ring to the inner ring is breakproof since it would involve outer ring mechanisms to operate.

To solve this problem the procedure that does the signalling in the outer ring can be placed in the inner ring (the protected environment). It would then be able to signal conditions normally on the outer ring stack and then switch over to the inner ring stack if no handler were found or if there were an error while attempting to signal such as a misthreaded stack in the outer ring.

A final problem discovered in multi-ring signalling has to do with an outward call followed by an inward call. In this case there are two outstanding invocations of the inner ring. Now, if the second invocation of the inner ring were a subprocedure of the first, the second invocation might depend on condition handlers in its parent procedure. However, in the normal PL/I signalling structure first the outer ring would have a chance of fielding an error in the second invocation of the inner ring before the expected handlers of the parent procedure in the inner ring would get control (if ever).

Using a debugger and figure 6.1 as an example, consider <A> to be the main procedure of the debugger and <A'> to be an internal procedure called by an activated breakpoint. Let <B> be the program that is being debugged. Now if <A'> signals a condition, <B> would have a chance to

field it, not allowing <A>, the main debugger program, to properly handle it. This example is somewhat contrived and may not seem realistic enough to the reader. However, programs are written with internal procedures and the error handling may be expected to work this way. Because of the lexical proximity of the internal procedures, the programmer might not consider the problem discussed. Again the solution is complete specification of error handlers even in subprocedures, and elimination of the "dynamic descendence" rule.

Chapter Seven

Implementation, Conclusions, and Future Research

7.1 Implementation

An implementation of some of the ideas in this thesis was undertaken to show that 1) the user environment can be partitioned in the manner described, 2) all the interactions between the environments were identified, and 3) rings are efficient for this separation. Multics was chosen as the system on which to implement the test environment because it supports the process model described in chapter two and suffers from the problems described in chapters one and two. Furthermore, Multics has rings implemented in hardware which would undoubtedly help make the implementation efficient. Finally, Multics was an easily available system to experiment on.

Chapter four discusses guidelines for support routine coding that facilitate their separation and protection. Those guidelines are basically modular design providing functional abstractions and data hiding. Experience with the test implementation reinforces the belief that protection would be simpler for those routines that followed the guidelines suggested, and harder for those that did not. In particular, both the event manager and timer manager (discussed in chapter four) were designed and coded as functional abstractions. Thus, by merely

assigning those procedures to the lower user ring and writing simple transfer vectors as gates to the entrypoints, the event and timer mechanisms were protected.

On the other hand, the design of the I/O system programs did not provide for information hiding. This forced I/O access programs to know the exact layout of the control blocks and to manipulate the blocks directly because of the lack of functional abstractions. Although the ideas in the I/O system are good (streams with high level interactions) [Feiertag and Organick 71], the implementation made it impossible to protect any of the features of the I/O system. Not protecting the I/O system as a whole did not affect the "connection" between the user and the command and control environment, however. The attachment of the terminal was "owned" by the inner ring and thus could not be affected by user written programs.

With the modules identified, the protected environment was established and a scheme for making outward calls and subsequent inward returns was designed and implemented. The functionality at this point was that of the original Multics process; a user typed a command line and the specified programs were found, executed, and followed by a ready message. The only difference was that the specified command was executing in a ring of less privilege than the "normal" user ring, which might cause the program to trip over incorrect access problems. (19)

---

(19) Since users were basically not concerned with rings, and programs only ran in a single ring, access was usually granted only to ring 4

Incorrect access to the command and control environment programs was just what was desired, though.

Earlier I described how a user gets into the control environment. Briefly repeating it here, the user presses the break or attention key on the terminal and is then talking to the control environment. Examining how this is actually accomplished identifies a problem on Multics.

The break is noted by terminal control software in the supervisor. A process interrupt is generated which causes the computation in the user's process to cease, and a condition "quit" is signalled on the user's stack in the PL/I defined manner. (20) Usually, the only handler for the quit condition is a default handler called when no other handlers have been found, and the bottom of the stack is reached looking for one. At this point, the listener/command processor modules are called, essentially entering the command environment. The process interrupt acts just like an interrupt on other systems in the sense that control is torn away from the executing procedure and is transferred elsewhere. This process interrupt essentially implements the "stop" mechanism of the control environment. Some of the commands the user may type actually execute in the protected environment, as described earlier, and perform the other control environment functions.

(the default when setting access is to choose the current ring). Thus, initially, many access problems were encountered.

(20) See chapter six for details on signalling.

Because the control environment is in a different ring now, care must be exercised to be sure that the quit signal is directed to that ring. Unfortunately, in the current system, all signals, including those arising from process interrupts, are signalled on the stack of the current ring of execution. This means that when actually executing a user program, the quit signal would be first signalled on the user program stack, and then, if directed properly, would continue on the protected environment stack. Thus, a destroyed user program stack could prevent returning to the control environment forever!

Timers are implemented as process interrupts too. Thus an inner ring wishing to be notified at a certain time, or after a certain amount of chargeable execution time, would be subject to user stack integrity. This sort of dependency obviously violates ring structure and the goals set forward in chapters one and two.

Recently a proposal a been made to solve this problem. The solution is simply to poll inner rings first when a process interrupt condition is to be signalled. Thus an inner ring has "first crack" at handling these conditions and would allow user programs to handle them only if they were of no interest to the inner ring at that time.

The problem of user programs wanting to handle quits comes up again here. (21) The solution proposed in chapter five is useable here as well; the quit key can be multiplexed by some means and the proper

---

(21) This was discussed in chapter five.

process interrupt will be generated depending on the "severity" of the abort that the user wanted.

Error condition handling resulted in a study of error signalling mechanisms in detail. The results of the study are presented in chapter six. The implementation finessed some of the problems discussed in chapter six with special case code that handled the more common problems.

In summary, the implementation proved that the proposed separation could be done, and furthermore, was relatively easy given a certain good style of coding to deal with. Simple experiments indicated that the cost of using the implementation was approximately two to three times that of the original system when executing a program that did nothing more than return. For more complicated programs, the cost was essentially a fixed overhead (of two or three times the normal program invocation cost, as in the "nothing" program) which becomes insignificant when compared to a PL/I compilation, for example. The cost can be expected to decrease if more users were sharing the code, but the amount is not determinable.

7.2 Conclusions

This thesis shows how system designers, interested in a system with verifiable security properties, and users of system, interested in self-protection measures, can both be satisfied by a rather simple hardware mechanism that provides (at least) three protection environments. Digital Equipment Corporation included three protection environments in the PDP 11/45 (and extensions to it), but failed to provide an ordering for all three; one environment, "kernel" mode, was given usual supervisory privileges but the remaining two were left unordered. This thesis discusses why the ordering of privileges is needed and useful to facilitate the establishment of a program development environment. In addition, this thesis shows that rings are indeed useful, and suggests that designers should consider including a ring-like mechanism in new systems.

7.3 Areas for Future Research

Many of the concepts discussed in this thesis constitute areas needing more research. Primarily, the problems of the "ideal solution" require research and experience with domain oriented systems to determine how the various components of the user environment should be managed. It may turn out that to solve some of the problems pointed out in chapter two, such as the access required by a linker or debugger, rings may be needed.

Better high level languages with more intelligent compilers can help solve some of the problems of the programmer. More often than not it is the problem of representation of information that causes programmers to invent unclean techniques in their programming. CLU [Liskov 76] might help in this respect, allowing users to define extended type objects and procedures to manage them and preventing any other procedures from manipulating the internal structure of the extended type objects.

The programmer's apprentice concept [Hewitt] can be a very valuable aid to the programmer, but this seems years off. The concept of front end processors requires more research to decide the functionality and level of independence from the host required by the front end to make it suitable for use with differing machines, a collection of similar

machines (as in a network), and a combination of these two ideas. In the future we might encapsulate the complete command and control environment, discussed in chapter five, in a personal computer that would determine the resouces required for any command and dynamically acquire them from a network of resources.

Appendix A

Certification and Kernel Simplification

With the growing trend of entrusting computer based systems with important company information and/or finances, as well as computerized cash flow, it becomes important to many companies to have a guarantee or proof that the system will not malfunction nor produce erroneous results, because the system is now dealing with real dollars. Thus the concept of certifying a system came about which meant that someone was willing to guarantee that a system functions properly under all circumstances and will not allow unauthorized modes of access to the system or data. In particular, the system has to be shown to 1) not release information to unauthorized personnel, 2) not allow unauthorized modification of information, and 3) not allow one user to deny service to another. It is hard to prove these types of negative attributes about a computer system because modern systems are so large and have many complex transactions going on within them. Many researchers are working on developing methods that can automatically verify that a system performs as specified, but even the specification techniques have not yet been perfected. One problem with developing specifications is that all the types of interactions are not fully known or understood especially in a system which has some degree of undecidedness built into it (i.e. multiple processes and their scheduling). Automatic program verification techniques are still in their infancy and furthermore would

usually require rewriting the entire system in a new language with other constraints. Thus, the only alternative at this time is to review the code of the system manually and understand it fully so that it would then be possible to decide if the system is "secure".

An approach to ease the burden on a system certifier, or even make it possible, is to concentrate those programs dealing with security into a security kernel and leave all other functions outside. This approach enables a certifier to ignore all those programs outside the security kernel, and thus leaves behind a smaller amount of code to be examined. It is obvious that less code would be easier to review and comprehend.

Thus, any programs not dealing with the security of the system, as described by the three points mentioned above, should be removed from the supervisor. In keeping with this aim, Janson and Bratt have described how the dynamic linker and name space manager can be removed from the Multics supervisor [Janson 74, Bratt 75].

In the past, however, there was another reason why programs, which were primarily user programs, should not be in the supervisor. This reason was based on the "principle of least privilege" [Saltzer and Schroeder 75] when deciding proper placement of a module. This principle, stated simply, means that a program should only have as much access as it needs to do its job. Otherwise programming errors in one program may lead to the destruction or unrelated databases and other programs, which obviously is fatal in a supervisor. (22) This guideline

was followed as a simple rule of good design; but now, certification researchers realize that uneeded access makes it harder to certify a system correct because it must be shown that programs do not take advantage of extra privileges they might posess in addition to showing that they do their job correctly. It is not unusual to discover that good design principles also fit in well with certification work, as seen in this thesis.

As a result of the certification work, it became apparent that the user environment would fill up with modules that were previously protected, and now these programs would be subject to the same programming errors that harm other user programs. This loss of function due to the increased fragility of the user domain is what this thesis is about.

(22) In fact, this is the same problem that is being attacked in this thesis, only this time the effects are more serious than a lost user process.

BIBLIOGRAPHY

The M.I.T. Laboratory for Computer Science
was formerly known as Project MAC

[ARPANET] Roberts, L. G. and Wessler, B. D., "Computer Network Development to Achieve Resource Sharing", Proc. AFIPS SJCC, vol. 36, 1970, pp. 543-549.

[Bratt 75] Bratt, R. G., "Minimizing the Naming Facilities Requiring Protection in a Computing Utility", M.I.T. Laboratory for Computer Science Technical Report 156, Cambridge, Mass., September, 1975.

[Broughton] Broughton, J. M., "An Extensible Command Language for the Multics System", B.S. and M.S. thesis at the Massachusetts Institute of Technology, Cambridge, Mass., May, 1976.

[CLUnote 43] Snyder, A., "A Proposal for an Error Handling Mechanism", M.I.T. Laboratory for Computer Science Computation Structures Group, CLU Design Note #43, Cambridge, Mass., March 1975.

[CLUnote 60] Liskov, B., "Exception Handling", M.I.T. Laboratory for Computer Science Computation Structures Group, CLU Design Note #60, Cambridge, Mass., August, 1976.

[CTSS] The Compatible Time-Sharing System: A Programmer's Guide, M.I.T. Press, 1966.

[DCS] Rowe, L. A., "The Distributed Computing Operating System", University of California at Irvine, Department of Computer Science Technical Report #66, June, 1975.

[Denning 68] Denning, P. J., "The Working Set Model for Program Behavior", CACM 11, 5 (May 1968), pp. 323-333.

[Dennis 64] Dennis, J. B., "Program Structure in a Multi-Access Computer", M.I.T. Laboratory for Computer Science Technical Report 11, Cambridge, Mass., May, 1964.

[Dennis and Van Horn 65] Dennis, J. B., Van Horn, E. C., "Programming Semantics for Multiprogrammed Computations", M.I.T. Laboratory for Computer Science Technical Report 23, Cambridge, Mass., December, 1965.

[Feiertag and Organick 71] Feiertag, R. J. and Organick, E. I., "The Multics Input/Output System", ACM 3rd Symposium on Operating System Principles, Palo Alto, California, October, 1971, pp. 35-41.

[Goodenough 75] Goodenough, J. B., "Structured Exception Handling", Proceedings 2nd ACM Symposium on Principles of Programming Languages, January 20-22, 1975, pp. 204-224.

[Gould 75] Gould, I. H., "Interactive Debugging System for a Multiprogrammed Minicomputer", Interactive Systems, London, England, September, 1975 (Uxbridge, Middx., England: Online 75), pp. 451-464.

[Graham 68] Graham, R. M., "Protection in an Information Processing Utility", CACM 11, 5 (May 1968), pp. 365-369.

[Greenberg 76] Greenberg, B. S., Private communication.

[Hewitt] Hewitt, C., "Towards a Programming Apprentice", IEEE Transactions on Software Engineering SE-1, 1, March, 1975.

[Huber] Huber, A. R., "A Multi-Process Design of a Paging System", M.I.T. Laboratory for Computer Science Technical Report 171, Cambridge, Mass., December, 1976.

[IBMCP] IBM Virtual Machine Facility/370: CMS User's Guide (Release 3), International Business Machines Corp. Order no. GC20-1819, February, 1976.

[Janson 74] Janson, P. A., "Removing the Dynamic Linker From the Security Kernel of a Computing Utility", M.I.T. Laboratory for Computer Science Technical Report 132, Cambridge, Mass., June, 1974.

[Janson 76] Janson, P. A., "Using Type Extension to Organize Virtual Memory Mechanisms", M.I.T. Laboratory for Computer Science Technical Report 167, Cambridge, Mass., September, 1976.

[Jones 73] Jones, A. K., "Protection in Programmed Systems", Ph.D. Thesis, Carnegie-Mellon University, 1973.

[Liskov 72] Liskov, B., "A Design Methodology for Reliable Software Systems", Proc. AFIPS FJCC 41, 1972, pp. 191-199.

[Liskov 76] Liskov B., et al., "Abstract Mechanisms in CLU", M.I.T. Laboratory for Computer Science, Computation Structures Group Memo #144, Cambridge, Mass., April, 1976.

[Montgomery 76a] Montgomery, W. A., Private Communication.

[Montgomery 76b] Montgomery, W. A., "A Secure and Flexible Model of Process Initiation for a Computer Utility", M.I.T. Laboratory for Computer Science Technical Report 163, Cambridge, Mass., December, 1976.

[MPM] Multics Programmer's Manual - Reference Guide, Order No. AG91, Rev. 1, Honeywell Information Systems, Waltham, Mass., December, 1975.

[NSW] Crocker, S. D., "The National Software Works; A New Method for Providing Software Development Tools Using the ARPANET", Proc. Meeting on 20 Years of Computer Science, Pisa, Italy, July, 1975.

[Noble 69] Noble, J. M., "The Control of Exceptional Conditions in PL/1 Object Programs", Information Processing 68, North Holland Publishing Co., Amsterdam, 1969.

[Organick 72] Organick, E. I., The Multics System: An Examination of its Structure, M.I.T. Press, Cambridge, Mass., 1972.

[Organick 73] Organick, E. I., Computer System Organization: The B5700/B6700 Series, Academic Press, New York, 1973.

[Parnas 72a] Parnas, D. L., "A Technique for Software Module Specification with Examples", CACM 15, 12, pp. 1053-1058 (December 1972).

[Parnas 72b] Parnas, D. L., "Response to Detected Errors in Well Structured Programs", Carnegie-Mellon University Technical Report, July, 1972b.

[PSN25] Saltzer, J. H., Hastings, T. N. and Daley, R. C., "Unified Control of Enabled User Traps, Including Memory Protection and Relocation", M.I.T. Computer Center Programming Staff Note 25, May 27, 1964.

[PSN26] Hastings, T. N., "Requirements of the FAPBUG Program", M.I.T. Computer Center Progamming Staff Note 26, March 9, 1964.

[RCTE] Crocker, D. C. and Postel, J. B., "Remote Controlled Transmission and Echoing TELNET Option", Arpanet Protocol Handbook, Rev. 1, Network Information Center, Stanford Research Institute, Menlo Park, California, April, 1976.

[Redell 74] Redell, D. D., "Naming and Protection in Extendible Operating Systems", M.I.T. Laboratory for Computer Science Technical Report 140, Cambridge, Mass., November, 1974.

[Reed 76a] Reed, D. P., Private Communication.

[Reed 76b] Reed, D. P., "Process Multiplexing in a Layered Operating System", M.I.T. Laboratory for Computer Science Technical Report 164, Cambridge, Mass., June, 1976.

[Saltzer and Schroeder 75] Saltzer, J. H. and Schroeder, M. D., "The Protection of Information in Computer Systems", Proceedings of the IEEE 63, 9 (September 1975), pp. 1278-1308.

[Schroeder 72] Schroeder, M. D. "Cooperation of Mutually Suspicious Subsystems in a Computer Utility", M.I.T. Laboratory for Computer Science Technical Report 104, Cambridge, Mass., September, 1972.

[Schroeder 75] Schroeder, M. D.. "Engineering a Security Kernel for Multics", ACM 5th Symposium on Operating System Principles, Austin, Texas, November, 1975, pp. 25-32.

[Schroeder and Saltzer 72] Schroeder, M. D. and Saltzer, J. H., "A Hardware Architecture for Implementing Protection Rings", CACM 15, 3 (March 1972), pp. 157-170.

[TENEX] Bobrow, D. G., et al., "TENEX, A Paged Time Sharing System for the PDP-10", CACM 15, 3 (March 1972), pp. 135-143.

[TR123] Saltzer, J. H. ed., "Introduction to Multics", M.I.T. Laboratory for Computer Science Technical Report 123, Cambridge, Mass., February, 1974.

[Yates 62] Yates, J. E., "A Time Sharing System for the PDP-1 Computer", S.M. Thesis at the Massachusetts Institute of Technology, Cambridge, Mass., May, 1962.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 11/3/95

Report # LCS-TR-175

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 108 (108-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☒ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): 2,38,48,86, PAGE FOLLOWING 101

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP! (1-102) UNNUMBERED TITLE PAGE, 2-101, UNNUMBERED BLANK
(103-108) SCANCONTROL, COVER, SPINE, TRGT'S(3)

Scanning Agent Signoff:
Date Received: 11/3/95 Date Scanned: 12/1/95 Date Returned: 12/7/95

Scanning Agent Signature: Michael W Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 12/1/1995

M.I.T. Libraries
Document Services