M.I.T. Laboratory for Computer Science
Computer Systems Research Division

May 4, 1976
Request for Comments No. 111

Protecting User Environments

Harold Jeff Goldberg

Attached is my recently accepted thesis proposal.

Massachusetts Institute of Technology

Project MAC

Computer Systems Research Division

Cambridge, Massachusetts

Proposal for Thesis Research in Partial Fulfillment

of the Requirements for the Degree of

Master of Science

Title: Protecting User Environments

Submitted by: Harold J. Goldberg
60 Wadsworth Street Apt. 25E
Cambridge, Massachusetts 02142

Signature of Author

Date of Submission: April 29, 1976

Expected Date of Completion: October 1976

Statement of the Problem:

Faulty programs have been known to destroy the environment in which they are run, causing those and other programs to malfunction. Although much system design effort has gone into protecting users from each other, not much attention has been paid to protecting users from their own programs. A method is proposed to solve some of the problems faced, using the concept of rings of protection. The proposed solution has the added advantages of not adding complexity to the supervisor of the system, nor increasing the amount of code that is concerned with the interaction between different users.

Supervision Agreement:

The program outlined in this proposal is adequate for a Master of Science thesis. The supplies and facilities required are available, and I am willing to supervise the research and evaluate the thesis report.

David D. Clark,
Research Associate in Electrical Engineering
and Computer Science

I. Introduction

In today's computer systems, programs are usually run in an execution environment (EE). The EE is an abstract machine based on the actual hardware, enhanced by system and user software. The EE contains those support routines that implement objects needed, but not created by the programmer, such as I/O packages, floating point support, stack management, among many others. A problem arises when programs have direct access to the EE, since they then have the ability to damage or destroy it. This is undesirable for several reasons. There is usually some overhead in etablishing the user environment such as the creation of a job or process, and the linking and/or loading of program modules. Therefore one does not always wish to consider the EE readily disposable. Furthermore, there may be state information maintained in the EE (such as open file information) whose loss might either destroy, or cause inconsistencies in important data. Then simple programming errors might unintentionally destroy this state information, causing the destruction of unrelated data bases. Debugging programs becomes harder because unnoticed errors may show up much later in the form of malfunctioning "tested and debugged" programs. To prevent these kinds of problems, some form of error detection and "clean abort" mechanism is desirable so that errors may be caught as soon as they occur and allow analysis of the situation.

It is now clear why protection of the execution environment is desirable. In this paper a solution to some of the problems faced is proposed. A method for the protection of the execution environment is suggested and is extended to encapsulate a control and command environment as well. This proposal is compared to other methods in use today. A scheme for how this proposal could be implemented on Multics is discussed and is followed by a proposal for actually implementing key ideas as a test case.

II. Protection of the Execution Environment

As pointed out in the introduction, a method of achieving the goal of invulerability of the EE and ease in debugging faulty programs would be to protect the EE from the executing programs. This means that the program and its environment must be separated by one of the protection means available at the present. Methods of separation/protection considered here are processes, supervisor implantation, and rings.

Processes are a good means of implementing separation and protection, and have the useful property of parallel execution which may result in increased speed. However, interprocess communication techniques are inherently slow and/or expensive since they require process scheduling or "busy waiting". Theoretically processes may be an excellent means of separation, but for real systems they incur too much overhead for the high bandwidth connection needed between a program and its environment.

[illegible]

Code for the EE can be placed in the supervisor of a system to protect it as the system code is protected from users. But this is a poor idea for protecting an execution environment for at least three reasons. By placing code in the supervisor, it becomes much harder to fix any programming errors that arise in the EE because each time one is found a new system must be installed. Furthermore, if a particular choice of options or implementation of the EE is not liked by some, they must either be dissatisfied and suffer, or some additional complex machinery must be added to the original code. Placing code in the supervisor usually annoints that code with special powers to reference and/or modify data that the program is not concerned with. This allows every program that is brought into the supervisor to inadvertantly cause unauthorized release of modification of information of users or protected parts of the system. Then each program would have to be carefully checked to be sure that it did not reference unrelated objects. Allowing programs to have more power than they need to do their job is the complete opposite of a desired feature of programs, which is to follow the "principle of least privilege". Moreover, supervisor implantation increases the size and complexity of the supervisor. Interestingly enough, it is all the problems with supervisor implantation that current research is trying to eliminate [Schroeder 75]. However, code in the supervisor is easily called by user programs and does have the advantage of direct memory access to user areas. Thus high bandwidth communication between user programs and the supervisor is possible.

Since the direct calling and memory referencing are desirable features of supervisor implantation (besides the obvious protection feature), perhaps a method should be chosen that has the advantages but not the drawbacks of such an approach. The concept of rings of protection, or simply rings, is what is desired here. Rings are just an extension to the user/supervisor modes with any number of levels of protection. A ring of level j has unlimited access to rings of levels j + 1 through the maximum ring number, but only controlled access (such as call or no access) to rings of levels 0 through j - 1. Using this concept the supervisor can be encapsulated in the lowest rings and have user programs and their environments in the higher rings. Then the EE may be encapsulated in the lowest of user rings with programs running in rings higher than that. In this way the EE is not part of the supervisor but is protected from the user programs in the same way that the supervisor is. Thus, it is possible to stop immediately upon the detection of a bad reference to the EE in the same way that bad references to the supervisor are detected in other systems. Thus a clean abort can be obtained, and analysis of the situation -as it occured- is possible. One should now see that the concept of rings is ideal for the protection of the EE.

III. Extension of Protection to Control and Command Environments

The command environment is the environment the user is in communication with immediately after having been assigned a process. The user instructs the command environment to run certain programs or perform tasks, via some command language which may have been tailored to that user's needs.

The control environment enables a user to stop and/or destroy a computation. The practical reason for letting a user have this power is simply economics; a looping program must be stopped, or it will use up limitless amounts of chargeable resources. It is also logically unsound to continue a useless computation. Destroying a process is really an extreme measure and is not what is usally wanted. Users would rather want to stop and examine a process and then possibly debug then continue them (dynamic debugging).

The interest that this research has in the control environment is to determine how, and at what level this process stopping should be implemented. Clearly, with perfect, functioning user code the stopping could be implemented directly in the user program environment. However, the point is that there is never perfect code at all times. Thus, the control environment, responsible for stopping a process, must be protected from damage by user programs. Hence, a choice of how to protect the control environment must be made.

Before protection is discussed, it would be wise to consider how one would like to use the control environment. Clearly some signal is sent from the user to the program which causes the process to stop. But then the user must communicate with some environment to tell it what the process should do next. A user might ask to enter "debug" mode, allowing memory dumping and patching. The user may then say "continue" or "re-initialize". More important, for this discussion, the user might also wish to first run some other system command or user written program. In this way the user could request help from a friend by sending a message, or determine if the stopped program has been functioning properly to that point by use of a calculator program. The way that the user requests these thing to happen is by issuing commands to the control environment. However, presumably the user already has a different way of issuing general commands -the command environment- which, in general, is tailored to that user's needs. For instance, the command environment may contain a list of abbreviations the user likes. The point here is that since the user already has a command environment, why complicate the user interface by making two -one for control environment commands, and the other for "regular" commands.

Since the control environment will be considered merged with the command environment, the command interpreter must identify control commands as well. The command environment may present an entirely different interface to the system than other users see. To accomplish this, the command environment must be extensible, in some sense, to accomodate each user's requirements. In general it should be user replaceable. These requirements are most easily satisfied by placing the command environment in the user program environment. Unfortunately, this once again allows users to harm themselves because their programs can damage the command environment. So once again protection is required for something that is primarily a user program.

The preceding idea of protecting a programs's execution environment is applicable to both batch oriented and timesharing systems alike, but the concept of control and command environments is generally restricted to timesharing systems. In batch systems the only type of control allowed might be specified in terms of maximum resource usage to allow to a given job. The command language might allow testing of return codes to determine execution paths, but is limited to waiting for the termination of each of the tasks to detemine the value of the return code. Although the remainder of this paper will be concerned with the environments found mainly on timesharing systems, a definite anlogy can be made between the command environment that will be discussed and the "JCL" interpreter found on batch systems. Thus, one might apply these suggestions, in a limited way, to batch systems too.

Because it is desirable to protect both the control and command environments, and the control environment essentially depends on the command environment, this research will consider the two environments as one unified control/command environment (UCCE). Hence the following discussion on protection considers the UCCE as a whole.

Immediately one should see that supervisor implantation is inapropriate for protecting the UCCE since that results in non-modifiable code. The the hopes of having the versatility described is lost. Even so, TENEX [Bobrow] uses this approach to protect its command environment. However, users of TENEX clearly see the differences between command that they write, and system commands, in things such as 1) passing arguments to programs by typing them on the same line as the command only works for system commands, and 2) having reserved names for all system commands.

Now here is a good opportunity to utilize processes for protection since high bandwidth is not important. Placing the UCCE in a separate process from the running user program also helps debugging by causing less disturbance in the user program environment. Parallel execution is also desirable, allowing user to UCCE communications to go on independent of, and simultaneously with user program execution. This feature facilitates monitoring of execution. There are drawbacks to using proceses, however. Depending on what system is being used, one process may not always be able to directly reference the address space of another. Then dynamic debugging is hard, if not impossible. Similarly, there may be no means for loading a program into the other process. If interprocess communication is used, in any form, then that part of the communication path which lies on the receiving side (the user program side) becomes vulnerable to user program destruction, and nothing has been solved.

Even if the described problems with processes are not problems in the system in question, it is still not necessary to use processes for this protection. Economy of mechanism may be achieved by reusing a mechanism that has been made available previously. Furthermore, current technology has not yet reached a point where processes are cheap enough to use in all situations where they are conceptually elegant.

It has already been shown how one may protect what may be considered user code, from users. The solution chosen was to use rings. That solution may also be applied here and thus no new mechanism need be introduced to the total solution. The structure proposed is to simply encapsulate the UCCE in a user ring which is lower than the ring in which user's programs execute. Once again, the code is protected from the user but is still not part of the supervisor. Dynamic debugging is easy since the UCCE in the lower ring can directly reference and/or modify the user programs in the higher ring. More important, however, is the fact that the user then has an environment which is known to always be functioning even when all of the user's programs may have been destroyed. In this way the user can maintain absolute control of a process.

The reader should notice that protecting the UCCE is indeed distinct from protecting the EE since in either case the other was not mentioned. On the other hand, each environment might benefit by protecting the other from it. For example, the UCCE is made up of programs which require some environment to run in. Thus protecting the UCCE's EE might uncover some unknown problems and prevent catastrophic errors due to a misprogrammed UCCE. In the other direction, debugging the EE can only be accomplished by providing an environment from which to debug it in. Because of the possible infinite recursion that may occur here, this research intends to encapsulate both the UCCE and the EE in one ring for simplicity. However, the code is hoped to be designed in such a fashion as to be independent of what ring it executes in. Then a cautious user may, in that user's own process, contruct any number of layers of EEs and UCCEs.

IV. Initial Thoughts on Implementation

Implementation of this proposal on the Multics system would provide an excellent opportunity to demonstrate the feasibility of the proposed design. This is true because Multics 1) is a process oriented timesharing system, 2) has rings implemented in hardware [Schroeder 71], 3) suffers from the problem of both UCCE and EE vulnerability, and 4) is a readily available system to the author. In this section initial thoughts on implementation on Multics will be presented. These should not be considered final design decisions by any means. They are simply thoughts on how this research may proceed. The reader is cautioned that some Multics specific terms and concepts are included which may decrease the readability of this section to the uninitiated.

The UCCE is discussed first.

The modules containing the command processor can easily be isolated but one must be warned that any "abbrev" processors and ready message programs are part of the command environment. The command processor will have to be modified to call programs in a higher ring. The user's terminal will have to be attached in the ring of the command processor (or in a lower ring, but it would serve no special purpose there), requiring that a special cross-ring I/O module be written for use by all user programs. The methods of entry into the UCCE will be fourfold: 1) via a call during initial process startup, 2) by user programs returning to their caller after completion, 3) by any abnormal conditions detected in the user environment, and 4) via the "quit" mechanism.

The last of these will alllow process stopping at any point by the user when the "attention" or "break" key on the control terminal is pressed.

In the remainder of this section a few of the common mechanisms in the user environment will be discussed, along with a description of how they may be protected, if at all.

The stack is one major common mechanism desiring protection. The protection of stack threads and return pointers would be useful, as would the limiting of access only to frames that a procedure should know about (display pointers). Such features are found on Burroughs [Organick] computers, and are implemented in hardware. Since Multics lacks this hardware, a recompilation of every program in the system would be required so that all programs would make use of some new software that would accomplish the same thing. Then each stack frame could exist in a separate segment, and have access to the frame set accordingly. This is obviously not a practical solution both for actual execution time and for conversion overhead.

On Multics, the stack is a repository for other distinct objects such as the condition chain, used for setting up condition handlers, the Linkage Offset Table (LOT), used for finding pointers to programs linkage sections, and the Internal Static Offset Table (ISOT), used for finding pointers to internal static sections ("own" variables) of programs. The "base of the stack" has many special pointers that define the user environment, such as pointers to the LOT, ISOT, signalling procedure, and special operator pointers (call/return). Again, it would be desirable to protect all of these objects, but most would require enormous system-wide conversions. The exceptions are the LOT and ISOT which could easily be protected by placing them in an inner ring and having them only readable from the user ring.

Dynamic linking on Multics currently incorporates many features that are conceptually not part of the linking process. Phillipe Janson describes this in his thesis [Janson 74] and a more specific paper [Janson 75]. The protection of the linker, once it is removed from the supervisor, is possible, and desirable, by using an inner ring to house it. It is clear that this can be done since it originally ran in a ring which was lower than the object ring. However, ring 0 must be made aware of the fact that the linker exists in a ring different from the ring where the linkage fault to be handled occurs. A major implication of having the linker in an inner ring is that it makes use of the reference name facility, which must therefore reside in the same ring as the linker or in a lower ring. Protection of the reference name facility, one it is removed from the supervisor, is also desirable, and so is compatible with this requirement.

Linkage sections are those parts of programs that contain impure data (modified during execution). Currentlty, linkage sections contain both linkage information as well as internal static (own) variables. Protection of internal static variables is not considered, since programs must be able to write as well as read them. However, the protection of linkage data is possible, and a reasonable goal. To accomplish this protection of linkage information, it must be separated from the internal static variables and protected. This separation is in fact being introduced as a result of work on the Multics pre-linking project, where large subsystems are to be pre-linked for faster execution. Once linkage and internal static sections are separated, the protection is easily accomplished by placing the linkage sections in an inner ring where the linker can write them, but where user programs can only read or transfer through them. Although it was mentioned that internal static variables could not be protected, the management of the area where they exist could be. [illegible] ?

The connection of I/O streams is a feature that should be protected, but due to the current structuring of the I/O subsystem on Multics, it would be impossible. Currently, the attachments are placed in I/O control blocks (IOCBs) which must be directly writeable by I/O modules. A future subsystem might restrict this and only allow modification to the IOCBs by calling subroutines. Then the protection of the I/O data should be reconsidered. It is important to point out that although I/O streams will not be protected, the attachment of the user's terminal will be in the UCCE. Thus the connection to the UCCE cannot be broken by faulty programs.

V. Proposed Work

The first step in achieving the desired protected environment will be to set up a protected quit handler/command environment with an I/O attachment to the user's terminal. The cross-ring I/O scheme will have to be developed. The test implementation will have the user's environment in ring 5, with the protected environment in ring 4. As mentioned earlier, the independence of what actual rings are used is desired, and the implementation will be programmed accordingly.

Once the protected command environment is functioning, the major portion of work for this thesis will begin. Modules will be brought into the protected environment in an orderly fashion, hopefully one function at a time. The exact order in which the work will proceed has not been finalized, nor has the total amount of time and effort to be spent on implementation been decided.

The procedure that is expected to be followed will be to identify a function or feature of the user environment that would benefit from protection. Then the protection of that feature, using the proposed design, will be considered. If it is obvious that the feature cannot easily be protected, as in the case of the stack mentioned earlier, a discussion of what other method may be used, or what modifications would be necessary for protection will be presented in the thesis. If the protection of that feature is considered trivial and not necesary for a working implementation, then it will only be implemetated if time permits. Similarly, if protection of a feature is possible but would require an enourmous amount of work, then perhaps only a fraction of that feature (the basic or important ideas) will be implemented, but a full discussion will be included in the thesis.

Although specific items have been pointed out in this paper, not all features of the user environment have been discussed. Other items that are candidates for protection are IPC data bases, timer management data bases, and a possible multi-tasking environment.

VI. Conclusion

Although protecting the execution environment solves some self-protection problems, others could be solved by setting up appropriate conventions for programmers to follow. One such convention might be that editors should give themselves write permission to a file only while they are actually writing it, as the Multics debuggers and compilers do.

One important point discovered by this work is the difference between the protection of the control, command and execution environments. The interesting conclusion is that all three can be protected by the use of rings. This approach provides the protection desired, and in addition, does not add complexity to the supervisor. Since their are not shared data bases between users, the code need not be certified for inter-user protection violations.

In conclusion, this work not only presents a solution to the "self protection" problem, it shows that rings are indeed useful for protection of code other than the supervisor, and thus should be available to users. This research also intends to determine, once and for all, the correct methods of handling multi-ring stacks and the proper approach to view multi-ring signalling.

References

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

[Janson 74] Janson, P. A., Removing the Dynamic Linker from the Security Kernel of a Computing Utility, M.I.T. Project MAC Technical Report TR-132, Cambridge, Mass., June 1974.

[Janson 75] Janson, P. A., Dynamic Linking and Environment Initialization in a Multi-Domain Computation, ACM 5th Symp. on Operating Sys. Principles, Austin, Texas, November 1975, pp. 43-50.

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

[Schroeder 71] Schroeder, M. D., and Saltzer, J. H., A Hardware Architecture for Implementing Protection Rings, CACM 15, 3 (March 1972), pp. 157-170.

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