PROTECTION IN AN INFORMATION PROCESSING UTILITY

Robert M. Graham
Massachusetts Institute of Technology
Cambridge, Massachusetts

Summary

One of the critical problems in the design of an information processing utility which permits flexible sharing of user information is that of privacy. This paper discusses one solution for this problem.

Introduction

In this paper we will define and discuss a solution to some of the problems concerned with protection and security in an information processing utility. This paper is not intended to be an exhaustive study of all aspects of protection in such a system. Instead, we concentrate our attention on the problems of protecting both user and system information (procedures and data) during the execution of a process. We will give special attention to this problem when shared procedures and data are permitted.

We will first give a brief resume' of those properties of an information processing utility which make protection necessary and non-trivial to implement. After a discussion of the desirability and necessity of protection we define a number of properties we feel are essential to any satisfactory protection scheme. We then describe an abstract model of the typical hardware used today for an information processing utility, and augment this model with an additional feature necessary for a satisfactory solution to the protection problem. Using this model we describe the properties required of the companion software. Lastly, we highlight certain additional complexities forced into the implementation of this protection scheme due to permitting shared information in a multi-processor system.

The Environment

The characteristics and properties of an information processing utility have been described in considerable detail elsewhere, the most comprehensive being Corbato' and Vyssotsky.[1] We will touch only on those properties which are pertinent to the problems of protection during execution. An information processing utility (IPU) will have a large community of users, many of whom are using the system simultaneously. The system will, of course, operate in a multi-programming mode and have more than one central processor. The community of users will certainly have diverse interests; in fact, it will probably include users who are competitive commercially. The system will be used for many applications where sensitive data such as company payroll records, will need to be stored in the system. On the other hand, there will be users in the community who wish to share with each other data and procedures. There will even be groups of users working cooperatively on the same project. Service bureaus, software producing companies, and other service organizations will have procedures which they wish to rent. Some groups may rent access to data bases. Finally, there will be public libraries of procedures supplied by the information processing utility management. Indeed, a primary goal of such a system is to provide flexible, but controlled access by a number of different users to shared data and procedures.

Why Protection?

Although protection is not necessary for privacy reasons in the case of a single user with his own private machine, it is certainly desirable. Protection in this situation aids debugging by limiting the propagation of errors, thus localizing the source of the original error. Even in fully debugged programs protection minimizes the effects of a human mishap or a machine malfunction. As soon as the machine is shared among more than one user, even if only one user at a time uses the system, protection is required so that the management may guarantee the highest possible reliability of operations as well as equity in charges to the users. For example, even the simplest multi-user system contains at least one data base which is shared by all users, namely the supervisor program itself. In addition, most contain information maintained by the supervisor regarding the allocation of resources and a record of resource usage for the purpose of charging users. Even though this data base may be used only via the supervisor, it is nevertheless shared by all users and so must be protected. Without adequate protection, a dishonest user might alter the accounting procedures or data thereby causing inequitable charges. A malicious user might even alter the system itself, causing it to act in an unreliable or destructive fashion. As soon as more than one user may have information stored in

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

the system at the same time, as in an information processing utility where the users store many files of information within the system for long periods of time, a user's privacy must be assured by the system and protection becomes even more critical. Without adequate protection in an IPU, a clever user, perhaps due to a single break or loophole in the privacy machinery, may be able to snoop in a competitor's data files obtaining information which gives him some material advantage. Such snooping would be difficult, if not impossible, to detect.

Properties of a Satisfactory Protection Mechanism

Excluding the running of all programs interpretively, any effective protection scheme must have some hardware assistance. In the past, the common hardware features for protection have been a mode switch for instruction execution and a memory bounds register. The mode switch specifies one of two modes of execution: master or slave. In mastermode any instruction may be executed, including a subset of the instructions called the privileged instructions. In slave mode an attempt to execute a privileged instruction causes a fault. The privileged instructions include the input and output instructions as well as the instructions for changing the mode switch and the memory bounds register. This effectively blocks users from accessing information written on the various storage media thus protecting inactive information in the system. Use of the mode switch alone does not protect information which is active and resident in working memory. This is the function of the memory bounds register. In partitions the working memory into two parts, one of which may not be accessed when executing in slave mode. Protection based on this type of hardware feature is an all-or-nothing type solution. If a program has any privileges it has all. If a program has any access, it has completely unrestricted access. We feel that this is unsatisfactory as a protection mechanism for an IPU. In a system where users may share data in working memory the ability to have more control over access is essential. What is needed is the ability to have a variety of access rights for each separate logical block of information (called a segment). Current machines which have been modified for use in IPUs have hardware features which allow memory to be subdivided into a large number of parts called segments. Each segment has a number of access control switches which specify various access privileges such as write/no-write, slave/master, and execute/no-execute. This hardware extension makes possible varying degrees of access to each segment which may differ from segment to segment. If this control is a part of the physical subdivision of memory, any user who has access at all, has the same access as every other user who may access the segment. The owner of a data segment needs write access if he is to maintain or update the segment with more timely information, but on the other hand it is necessary that other users of the data not be able to change it. Rather than being an exception, this is the rule in IPU. Thus the most advanced machines have hardware features such that the access may be varied on the logical segment rather than the physical segment, thus permitting different access by different users to the same physical segment.

In the design of a protection mechanism, an excellent guiding principle is the military security principle of "need to know". Applying this in the design of a protection mechanism in an IPU results in the property that each procedure has the minimum access needed to get its job done. A procedure has access to only those procedures and data segments necessary to do its task, and then only the type of access required for the job. This can be visualized by recalling the military system of clearances. The higher the clearance, the more documents one may access. On the other hand, the higher the clearance, the fewer the individuals that hold such a clearance. In an IPU the critical functions (i.e., those whose failure have disasterous consequences effecting the entire system) are segregated to the most protected area. This type of protection further improves the reliability of the system from that of the earlier two mode systems by further minimizing the extent of damage caused by hardware or software failure. Further, it aids maintenance. An IPU is a real-time system and the behavior of real-time systems is difficult, if not impossible, to repeat. The more compartmentalization and protection present in the system, the easier it is to isolate and locate the source of unwanted behavior. If the system has a general facility for layers of protection (analogous to layers of security clearance) then this service can be extended to users of the system. This permits restricted classes of users who use subsystems supplied by other users. A subsystem which has been designed and implemented by a user may then enjoy the same sort of layered protection with respect to its users as the operating system enjoys with its users. Such a service can be achieved easily by any user without any special administrative procedures on the part of the system management or any special coding by the user. Two noteworthy examples of the usefulness of such a service are a subsystem designed by an instructor for use by his students and a service bureau selling access to a specialized system.

In summary, a satisfactory protection mechanism should have the following properties. It should be possible to completely isolate one process from another; that is, a user should be able to deny any access whatsoever by other users to all of his segments. On the other hand, it should be easy and convenient for a user to allow controlled access to any of his segments, with different access privileges for different users. Further, within a single process layers of protection should be available for use by both the

system and a user so that the "need to know" philosophy can be applied to any degree deemed reasonable. Finally, it is extremely desirable that procedures may be called across the layers of protection without any special programming on the part of the calling procedure. If the grouping of procedures into protection layers is not coded into the procedure this organization is easily changeable by an administrative program.

The Abstract Model

In this section we will describe an abstract model of hardware features which will permit a satisfactory solution to the protection problems described earlier. This solution is but one of possibly several solutions of the general problem. It will illustrate each of the properties we consider essential to any satisfactory solution. We begin by describing a model which is essentially that of Dennis.[2] A key component of this model is a segment. A segment is a contiguous block of words whose length may vary during the execution of a process. Hardware for realizing segments is often called segment addressing hardware. Most computers suitable for use in an IPU also have paging hardware. While a segment is a logical unit of information of which a user is cognizant, a page is a unit of information which is useful to the system for storage management and is thus invisible to the user. Thus pages are not relevant to this discussion and will not be mentioned further. In a computer with segment addressing each word is addressed by an ordered pair of integers (S, W). S is the segment number and W is the word number within the segment. Segment numbers range from 0 to the maximum allowable number of segments in a process and the word number ranges from 0 to the current length of the segment to which it refers. Associated with each segment is a segment descriptor. The segment descriptor contains the absolute location of the beginning of the segment, the current size of the segment, and the access control indicator.

beginning of segment	length	access indicator

Descriptor

This access indicator specifies whether the segment may be accessed in slave mode, written, or executed. Further, if the segment is a procedure (i.e., execute indicator on), it specifies whether the procedure is to execute in master mode rather than in slave mode. Finally, it includes a fault bit which when non-zero, causes a fault (or trap or interrupt) on any attempt to reference the segment, even when in master mode.

If the write indicator is on but the execute indicator is off, the segment is writeable data. If the execute indicator is on and the write indicator off, the segment is a pure procedure (i.e., one which does not modify itself). If the slave indicator is on, any procedure may access the segment, otherwise only a master mode segment (one with the master indicator on in its descriptor) may access it. If the fault code is non-zero, no access at all is permitted. A non-zero fault code overrides the setting of all the other indicators. For every segment which a process may access (or potentially access), the corresponding descriptor resides in a distinguished segment called the descriptor segment. The segment number used in an address is, in fact, the index within the descriptor segment of the descriptor for that segment. In any system there will be a large number of descriptor segments, one for each process. Whenever a process is executing a hardware processor register called the descriptor base register contains the absolute location of the descriptor segment for the executing process. Thus, the contents of the descriptor base register indirectly define that set of segments to which an executing process has potential access.

In order to implement layered protection we augment the location counter and each descriptor with a field which will contain a ring number.

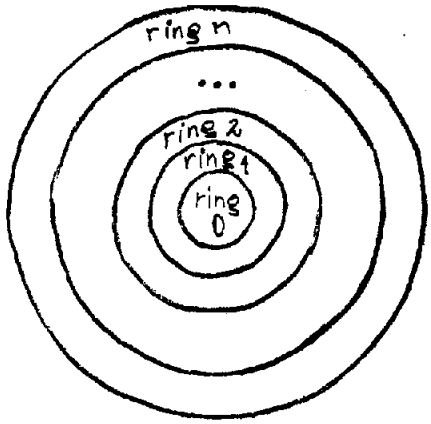
beginning of segment	length	access indicator	ring number

Descriptor

ring number	procedure segment number	word number

Location Counter

We define rings to be ordered, disjoint sets of segments, numbered from 0 to some maximum.



Each segment is assigned to one and only one ring. The lower the ring number a procedure is executing in, the greater its access privileges. A procedure executing in ring i has no access whatever to any segment in ring j, where $j < i$. On the other hand, a procedure executing in ring i has access to a segment in ring k if $k > i$, subject to the access restrictions specified by the indicators in its descriptor. However, to enforce this restriction, the system must be aware of the passage of control from one ring to another. In order to detect a "change of ring" in control, we further restrict the access rights of procedure segments. When a procedure executing in ring i attempts to transfer control to a procedure in any ring, other than ring i, a fault occurs. This fault is directed to the supervisor so that it may carry out appropriate housekeeping. We will discuss the kind of housekeeping necessary in some detail later in the paper. The assignment of each segment to a unique ring is sufficient to implement a solution of the protection problem. However, relaxing the disjointness requirement for the large class of chameleon-like shared service routines will result in a considerable increase in efficiency. This class of procedures need as much access privilege as their caller but no more. In this case the procedure will operate correctly in whatever ring control is in at the time it was called. Hence, we relax the condition that the rings are disjoint and allow a procedure segment to be assigned to a consecutive set of rings called its access bracket. The ring field of the descriptor will then contain two integers specifying the lowest ring and highest ring in the access bracket. Now a transfer by a procedure in ring i to a procedure with access bracket (n1,n2) with $n1 \leq i \leq n2$, does not cause a fault and does not cause control to change rings, i.e., control remains in ring i. This means the value of the ring field of the location counter does not change. A reasonable and useful interpretation of the access bracket can be made for data segments. Given a data segment, D, with access bracket (n1,n2), then a procedure in ring i may write into D if $i \leq n1$ (provided of course that the descriptor of D has the write indicator on), it may only read D if $n1 < ni \leq n2$ (even if the write indicator in D's descriptor is on), and may not access it at all if $i > n2$.

Software Support

The preceding section described a model of hardware features which will allow a satisfactory implementation of execution time protection. This section will describe the software support necessary to complete the job. We begin with the problem of permitting controlled entry into an inner ring from an outer (higher numbered) ring. Recall that when a procedure in ring i attempts to transfer to a procedure with access bracket (n1,n2) and $i > n2$ a fault occurs. Since it is usually undesirable to permit transfers to a procedure in an inner ring from all outer rings, we extend the notion of access bracket to include a third integer, n3, which defines the call bracket We will refer to the three integers as the ring bracket. If a procedure in ring i attempts to call a procedure, P, with ring bracket (n1,n2,n3) and $n2 < i \leq n3$ the call is allowed only to certain distinguished entry points in P. If $i > n3$ the call is not permitted at all. The call bracket is implemented by software. A fault occurs if $i > n2$ and the fault handler for this fault sorts out the case $n2 < i \leq n3$. One property of any segment with a non-empty call bracket is a list of entry points called the gate list. Because passing from one ring to another is similar to crossing a wall or fence separating the the rings, the entries are called gates and the fault handler which monitors the crossing is called the gatekeeper. A procedure in the call bracket may transfer only to those points listed in the gate list.

Before proceeding further, a few words about the origin of a segment's descriptor will help put our other comments in perspective, even though the origin of the descriptors is immaterial to the protection mechanism. In an IPU of the type we have in mind for a concrete realization of the solution described herein, the maintenance and storage of segments is entrusted to the file system. An inactive segment is stored in file memory and is often called a file. The symbolic name and other properties of the segment are kept in a file directory entry for the segment. The file directory entry contains the segment's symbolic name and its location in file memory. In addition, it contains the access bracket (n1, n2) and the call bracket (n3), the gate list, and the access control list. The access control list is a list of all users who may access the segment and the access which that user may have to the segment. A user is specified as a triplet (personal name, project id, process id). A three-part user specification makes it possible for the same person to have different access privileges when he works on different projects. By including a process id he may protect himself from himself by having different access in his various processes. When a user first attempts to reference a segment, by symbolic name, the file system locates the segment in file memory, assigns a segment number to it, and constructs the proper descriptor with access indicator depending upon what user is attempting to reference the segment.

We return now to the gatekeeper. When a procedure P in ring i tries to transfer to a procedure Q with ring bracket (n1,n2,n3) if $i < n1$ or $i > n2$ a fault occurs and the gatekeeper gets control. If $i > n3$ the transfer is rejected as an error. If $n3 \geq i > n2$ the target address of the transfer is compared with the entries on the gate list to validate that the call is to a valid entry point. If $i < n1$ the call is in an outward direction

and any entry point is valid. After the call has been validated the gatekeeper records, on a push-down stack, the return point corresponding to the call and the ring number of the ring control was in in at the time the call was attempted. An attempt to execute a return across rings also causes a fault which allows the gatekeeper to get control. The attempted return is validated against the record of unsatisfied calls across rings. This insures that returns remain in synchronization with calls.

The question of what ring control changes to when a procedure in ring i calls a procedure with access bracket (n1,n2) is still unresolved. The answer is that control should change by the smallest possible number of rings. Thus, if $i > n2$ control changes to ring n2, if $i < n1$ control changes to ring n1. This interpretation seems reasonable (although possibly arbitrary) for the following reasons. When entering an access bracket of lower numbered rings ($i > n2$) changing control to n2 adheres to the philosophy of granting only the minimum access necessary to do the job. In the other direction, changing control to n1 would grant enough access for the procedure to assist its caller, Q, if Q were called from a ring j where $n2 > j > n1$.

Additional Complexities in an IPU

The software support described in the preceeding section still does not satisfy the goals which we stated above. When a procedure in an inner ring is called arguments may be passed to the inner procedure. Argument lists include addresses. The inner ring having higher access privileges than the outer ring, may do damage to itself or other segments in its ring, inadvertently, if a calling procedure in an outer ring supplies the address of some segment in the inner ring. Thus, arguments passed to inner ring procedures must be validated, i.e., all addresses must be checked to see that the calling procedure actually was permitted access to the segments specified in the addresses. This is a standard operation and is a task that the gatekeeper can do for all calls to inner-ring procedures. This is not quite the entire story with regard to validation of addresses in argument lists. The fact that there are multi-processes executing on the same computer time-shared with other users, means that when segments are shared, data such as addresses in argument lists may change between the execution of two consecutive instructions. This is possible since the user may be interrupted due to a timer run out, for example, and another process may be executed before the interrupted process is resumed. If data segments are shared by these two processes, then validated argument list addresses may be modified by the interrupting process. One solution to this problem is to inhibit interrupts during the time the validation is taking place. Actually, one has to be considerably more sophisticated than this; interrupts must be inhibited until the called procedure in the inner ring is finished using the addresses. Even if such a long inhibiting of interrupts were tolerable, the problem is still not solved. In a multi-processor system, even if interrupts are inhibited, another process is executing on another processor. If that process is sharing a segment with this one, it will be able to modify the addresses during the time the inner procedure is executing. Thus, it is not enough just to validate the addresses. The addresses in the argument list must be copied into a data area which is in the same ring as the called procedure and this copy of the addresses is validated. This guarantees that the copy of the addresses which are being validated may be modified by a procedure in another process only if that procedure has access privileges which are equal to the called procedure in this process.

Another problem exists for calls in the other direction. When a procedure,P, calls a procedure, Q, in a higher numbered ring, the arguments P is passing to Q maybe in the same ring as P and thus inaccessable to Q. In this case they must be copied into a data area which is accessable by Q.

References

1. Corbató, F.J., and Vyssotsky, V.A., Introduction and Overview of the Multics System, Proceedings of the 1965 FJCC, Las Vegas, Nevada, November 1965, pp. 185-196.

2. Dennis, J.B., Segmentation and the Design of Multiprogrammed Computer Systems, JACM (October 1965), Vol. 12, No. 4, pp. 589-602.