Classification Codes: 161.E/0 and 1B2.2/0
Date: 5/18/77
Doct=9 Vers=1

The Protection of Information in a General Purpose Time-Sharing Environment.

Gary C. Pirkola and John W. Sanguinetti
The University of Michigan Computing Center
Ann Arbor, Michigan 48105

Introduction

The methods used to control access to information in the Michigan Terminal System (MTS) are described. Units of information to which access is controlled include both segments in the virtual address spaces of the various processes in the system and files in the on-line file system. The entities whose access to information is controlled are programs, at both the system and user levels, invoked by means of the command language.

In general, access to information is controlled in MTS in the following ways. First, the use of virtual storage in MTS is such that individual processes cannot refer to segments in the private virtual storage of any other process. Second, programs executing in a process in MTS will switch, under system control, between various domains (primarily between the user and system domains) and the access to segments in the process's private virtual storage will change depending on the currently active domain. Third, modes of access to individual files in MTS may be restricted not only to specific users and groups of users, but also to specific (user-written) programs. As a result, owners of files may completely determine under program control how their files may be accessed. Finally, users in MTS may, at the command language level, switch between various subsystems, and the access to a particular file may change depending on the currently active subsystem.

MTS

The Michigan Terminal System (MTS) is a large, general-purpose, time-sharing operating system which has been in continual development since 1966 by the University of Michigan's Computing Center staff, initially for use on the IBM 360/67. It is currently in use at the University of Michigan and five other universities in the United States, Canada, and England[1], running

on IBM 360/67, 370/168, and Amdahl 470V/6 computers. At the University of Michigan, MTS runs on a four megabyte Amdahl 470V/6 and typically supports 160 simultaneous users (both terminal and batch) during a normal afternoon. MTS services the educational and research computing needs of the University. In recent years, all of the universities using MTS have contributed in varying degrees to its development. In particular, the storage protection mechanism described in this paper was first implemented at Wayne State University.

MTS has been described in [1]. Only those aspects of the system's structure which are relevant to the protection of information will be described here. MTS is a virtual storage system in which each user has a process which has its own unique virtual address space. In the remainder of this paper, we will generally not distinguish between a user and the process which acts for the user. The principal component of the user's process is the main command interpreter. Each user process has a dedicated device -- usually a terminal -- for input of commands or data and output of command results and other messages. The command interpreter generally reads a command from the input device and performs the indicated operation. Some typical commands are SIGNON, SIGNOFF, RUN, EDIT, PERMIT, COPY, CREATE, and DESTROY. Each of the preceding commands, except for SIGNON and SIGNOFF, performs some operation on one or more files -- for example, the RUN command loads the object program found in the specified file and initiates execution.

The main command interpreter may invoke one of several command language subsystems (CLSs) in response to a user command. Each subsystem interprets its own set of commands. In this way, the main command language is kept to a reasonable size. For example, the context editor is invoked by the EDIT command, interprets editor commands until a STOP command is given, and returns to the main command interpreter. Although most CLSs appear to be part of the system, each is a separate set of routines which is invoked by the main command interpreter. In fact, a program which a user runs by means of the RUN command (hereafter called a user program) is just another subsystem to be invoked, as far as the main command interpreter is concerned.

One of the important properties of a subsystem is that it is independent of the other subsystems. A subsystem may be suspended, some other subsystem may be invoked, and then the original subsystem may be continued at a later time. Thus, a user could run a program, suspend it to edit an input file, and then continue running the program from the point of suspension. In fact, the main command interpreter can be considered a

---

[1] The University of Alberta, The University of British Columbia, The University of Newcastle-Upon-Tyne, Rensselaer Polytechnic Institute, and Wayne State University.

command language subsystem.

The main command interpreter not only interprets commands, it also provides system services by means of subroutines which user programs and CLSs can call. For instance, to dynamically acquire virtual storage, a user program calls the system-supplied subroutine called GETSPACE. All system services are provided in the form of subroutines -- user programs do not, in general, execute supervisor call instructions. System-supplied subroutines such as GETSPACE use supervisor calls to request services of the supervisor, which is the only component of MTS which operates in the "supervisor state" provided by the hardware of the 360/370-style machines [2]. Each user process operates exclusively in problem state.

Virtual Storage

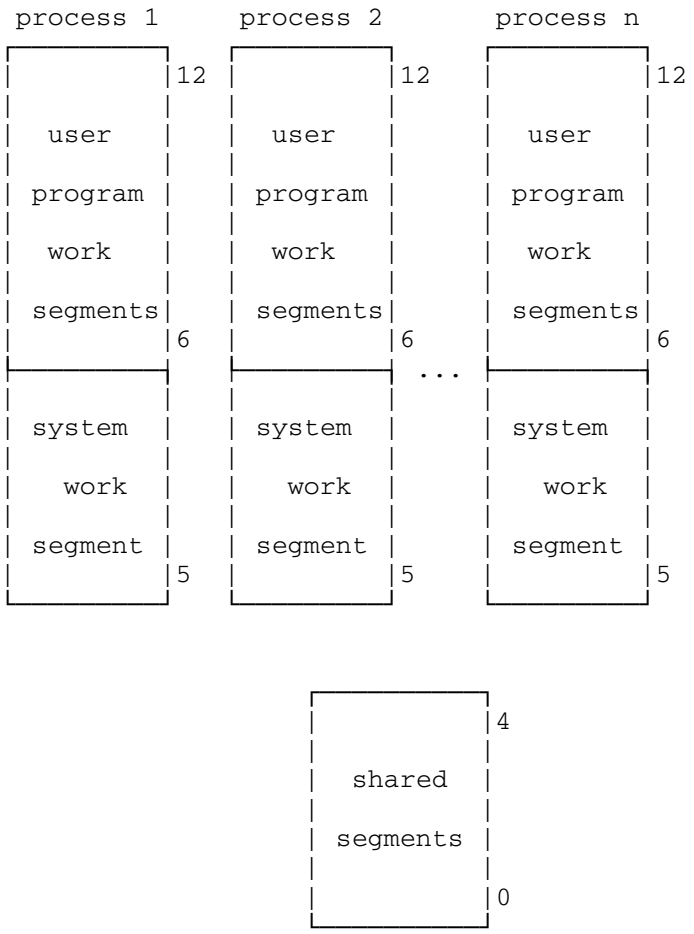
Use of Virtual Storage in MTS

MTS uses virtual storage in the following way (see figure 1). Each process in the system has its own virtual address space. This space currently consists of 13 segments of 256 pages each (segments 0-12) amounting to about 13 million bytes for each user. The first five segments (0-4) are shared among all processes and have the same addresses in all processes. This is the "address agreement" approach to shared memory. All of the shared system components reside in these shared segments -- the main command interpreter, most CLSs, the supervisor, the file system routines, and others. The remaining address space of each process (segments 5-12) is private to that process -- it does not intersect with the address space of any other process. Information in the private segments in any process cannot be accessed in any way by another process since that information does not appear in the address space of any other process [3]. This results from the use of the address translation mechanism and is the principal means of information protection between processes in MTS.

The private segments of each process are further divided into a segment for system work space (segment 5) and several user program work space segments (6-12). The system work segment contains all of the control information at the process level necessary to provide services to the user and to account for them.[2] No user-callable programs are ever kept in the system work segment. Among the control structures kept in the system work segment are many of those used by the file system and the CLSs. The user program work segments contain user programs

---

[2] System-wide control data structures, of necessity, are kept in the shared segments, but nearly all control structures which are specific to a single process are kept by the process itself in its system work segment.



Use of Virtual Storage in MTS

Figure 1

which are run and any work areas dynamically allocated by these user programs.

Protection of Information in Virtual Storage

As we have seen, the address space of each process can be divided into three parts -- shared segments, the system's work segment, and the user program's work segments. A security policy can be defined by putting different access restrictions

on each part. Because the IBM 360/370 architecture does not provide a hardware-implemented execute-only storage access type, every process must have read access to the shared segments in order to call system subroutines which reside there. Except for the relatively short time involved in executing a few system routines, a process should not have write access to the shared segments. System components must have both read and write access to the system's work segment, but user programs must have no access to it.[3] Both system components and user programs must have read and write access to the user program work segments.

In this discussion, the objects to which access must be controlled are segments. The subjects which access the objects are routines within a process -- that is, system subroutines, the command interpreter, the CLSs, the file routines, and user programs. Note that system-provided utility programs, like compilers, the linkage editor, etc., are all considered user programs, since they are invoked by means of the RUN command.

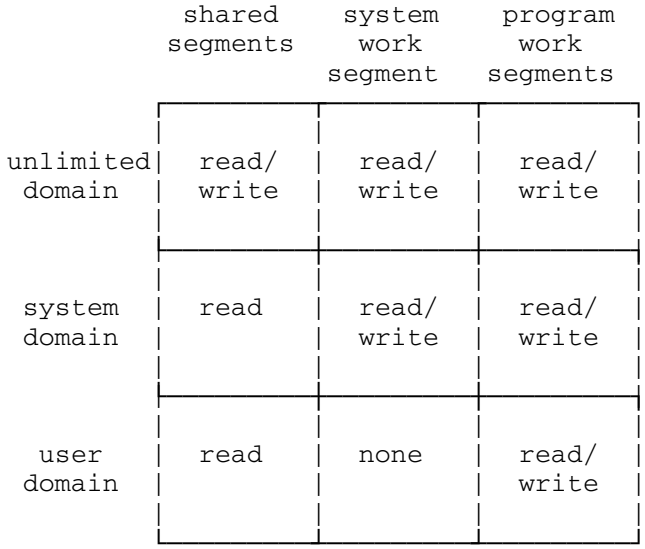
The desired access restrictions mentioned above define three separate "protection domains" within a process.[4] (See Linden [4] for a discussion of protection domains.) (1) Routines which require read/write access to all segments including the shared segments execute in the "unlimited" domain. For example, system routines which maintain information in tables located in shared segments must operate partly in the unlimited domain. (2) Routines which require read/write access to the system work segment, but not the shared segments, execute in the "system" domain. For example, the command interpreter operates in the system domain. (3) User programs require only read/write access to user work segments and read access to the shared segments (so that system subroutines may be called). This access defines the "user" domain. The security policy is implemented by controlling the transitions between protection domains. Figure 2 provides a summary of the access rights of domains to segments.

Control of Access to Virtual Storage

---

[3] In MTS, there is no "sensitive" information kept in the system work segment which must be hidden from the user. Therefore, user programs could be allowed read-only access to the system's work segment.

[4] The three protection domains described here are those which are relevent to a user's process. There are two other domains in MTS: absolute and supervisor. There are absolute processes which do not use virtual storage -- such as the paging drum processor. These processes execute in problem state with access to all of real storage. The supervisor itself executes in supervisor state with access to all virtual storage and all real storage.



Access to Segments by Domains

Figure 2

To provide the necessary access control, two separate mechanisms are used. To control access to shared segments, the storage key protection feature of the IBM 360/370 architecture is used. Pages in memory are given "store protected" storage keys, and the access to those pages is determined by comparing the key in the processor with the storage keys -- read/write access is allowed if the processor key matches the storage key; read access is allowed, write access is denied if the keys do not match. If the processor key is zero, then all pages are read/write accessible.[5] In MTS, pages in the shared segments are given storage keys of zero and pages in private segments are given storage keys of one. A user process always is given a processor key of one by the supervisor unless it is in unlimited domain. Thus, all subjects have read-only access to the shared segments.

Note that the use of storage keys to protect segments is

[5] See [2] for more details on how storage key protection works.

not especially appropriate, since keys are associated with pages on a 360/370, and not with segments. However, they can be used for this case of access control because the storage keys in the shared pages need never be changed once they are set initially, and because only very limited write access is allowed to the shared segments.

The same mechanism will not work to control access to the system's work segment. If the pages in the system work segment were given storage keys distinct from the keys given the pages in the user work segments, any subject which had read/write access to the system segment would not have write access to the user work segments (unless its processor key was zero, giving it write access to the shared segments as well).

Instead, the hardware-provided address translation mechanism (using the segment table) is used to provide different access rights to the various private segments. While system routines are executing, the segment table contains an entry for each of the shared, system, and user segments. While user routines are executing, the segment table contains entries for the user segments (and, of course, the shared segments) but not the system segment. This is the second use of the address translation mechanism for storage protection. A user process cannot ever access another process's private segments, and it can only access its own system work segment when it is in the system domain. The software must carefully control transitions between these two protection domains in order to insure that the segment table contains only the appropriate entries.

System-Controlled Domain Switching

A transition between system domain and unlimited domain is allowed, and is accomplished by a supervisor call. This is necessary, of course, because the supervisor must verify that the domain switch is, in fact, being requested from the system domain, and must change the process's processor key to zero if it is. Due to the fact that routines which execute in the system domain are "trusted", transition from system to unlimited domain is always allowed. It is presumed that the routine will switch its domain back to system domain as soon as it no longer needs write access to a shared segment.

Since segment tables are kept in the supervisor, transitions between system and user domains must be accomplished by means of supervisor calls. Two supervisor calls are defined for this purpose -- one to switch in each direction. Since domain switches must be controlled, these supervisor calls (particularly "enter system domain") must be allowed to be issued only from well-known system routines. In fact, since there is no need for a user program to issue any supervisor calls, all supervisor calls except "enter system domain" can be disallowed when in the user domain. All of the locations from which the "enter system domain" supervisor call can be validly

issued are known to the supervisor and are called "gates." All such gates are in the shared segments so their addresses remain fixed. Consequently, the supervisor always knows in which domain a process is executing and can assure that the system domain is only entered at pre-determined locations.

Implementation Considerations

In general, there is a gate for each non-trivial system subroutine.[6] The gate provides the domain change, and the subroutine can then access the system work segment and issue supervisor calls. Each system subroutine is required to return to its gate which executes the "enter user domain" supervisor call before returning to its caller.

There are several things which must be considered when allowing a subroutine in the system domain to be invoked by a user domain routine. The "attenuation of privilege" problem must be solved -- that is, the subroutine must not be able to be tricked into doing anything which its caller does not have suitable access to do. There are two cases where this can happen in MTS -- the caller could provide parameters to which he did not have access, or provide a save area whose address was in the system work segment. All system subroutines must check the addresses of the parameters and save areas they are passed to ensure that they are not in the system work segment. Note that this checking must be done after the domain switch but before the item being checked is used.

A less frequent situation which must be allowed for is the case of an "outward call" -- a routine in the system domain calling a subroutine in the user domain. In this case, any parameters or save areas provided to the user domain subroutine must be in one of the user segments. This usually involves making copies of parameters. A gate must also be provided through which the called subroutine may return. At the time of return, any returned parameters must be checked and any registers which are presumed to be restored must also be checked.

A routine executing in the system domain must not be allowed to lose control to a routine in the user domain due to an asynchronous or unexpected event, like a timer or attention interrupt. In MTS, this problem is handled by noting the occurrence of such an event if it occurs when the process is in the system domain, and exiting to the user domain trap routine (if there is one) at the time the domain switch from system to user domain is made. Those events which result in suspension of

---

[6] System-provided subroutines which do not access the system work segment can execute in the user domain, and thus do not require gates.

a system domain routine, like program interrupts or other errors, cause the main command interpreter to be invoked. If the user attempts to restart the suspended program, the domain is automatically switched to the user domain. This is required because the user, by issuing some intermediate sequence of commands, could cause the environment of the suspended system routine to change. Thus, most suspensions which do occur in system domain routines are errors, for which restarting is not allowed.

To summarize the storage protection in MTS, access to virtual storage is controlled on a segment basis, with three distinct protection domains: unlimited, system, and user. Between processes, non-intersecting virtual address spaces insure the isolation of private segments. Within a process, the differing access rights of the three domains protect the various segments. Transition may be made between the unlimited and system domains, and between the system and user domains. Because the unlimited and system domains are only available to system routines, which are presumed to be trustworthy, transitions between these two domains are not carefully controlled. On the other hand, transitions between system and user domains are very carefully controlled so that a routine in user domain has no direct access to the system work segment.

Limitations

Comparing this protection mechanism to other protection models, one finds that it is rather limited. If we ignore the distinction between the unlimited and system domains, this is a simple privileged state mechanism (or, if one prefers, a two-level ring structure [5]). In fact, the term "gate" comes from MULTICS. The segment table entries can be thought of as "capabilities" for the associated segments.[7] These capabilities are rather primitive, however, in that they allow only one type of access -- read/write.

The limitations of the protection mechanism are found primarily in its scope. The protection domains, as described by Linden [4], are not small, in the sense that system domain routines typically have more access than they need to perform their task. For instance, some system routines do not require any access to user segments, yet they have read/write access. One reason for this limitation is the lack of different access rights to user and system segments. By having an entry in the segment table, a segment is read and write accessible. Many system routines do not require write access to the system segment, for example, but there is no convenient way to provide them with read-only access. The other reason small domains are

---

[7] See [6] for a general description of capabilities and how they relate to the protection of information.

impractical in MTS is the number of available segments. Using 24-bit addressing, there are only 16 segments available in the virtual address space. This makes it impractical to load many different routines into many different segments which are members of different protection domains.

Some of these limitations could be removed by a simple change to the hardware. If the storage-protect key were put into the segment table entry rather than in each page, and if a hierarchy of access were defined for it (i.e. if a priority ordering were established) a 16-level ring structure could potentially be implemented. However, ring crossings would still be a high overhead operation, since they would require supervisor intervention.

Files

The File System in MTS

The file system is a collection of routines which are called either during interpretation of commands or while providing some service for a program, like reading or writing a file. Some of its primary responsibilities include (starting at the lowest level) initiating physical disk I/O and subsequent error recovery, allocating and deallocating disk storage space, and maintaining and interrogating the file system catalog. Moreover, it is a "buffer-oriented" system as opposed to a "virtual storage-oriented" system. That is to say, files in MTS are not mapped into segments of the process's virtual storage. Instead, "page-sized" buffers (as well as control blocks) are allocated in the process's system work segment for each active file within a process, and one of the main functions of the file system is to transfer blocks of file information, when necessary, between the page-sized physical records on the secondary storage device and the process's buffers. Quite obviously, since these page-sized blocks of information do not correspond directly to logical records read and written by programs, the file system is also responsible for managing and transferring logical records between the page-sized buffers and the calling program's input and output areas. Since each process has its own buffers and control blocks in its own private system work segment (and thus its own copy of a file during active use), the concurrent usage of files in MTS must be controlled by a set of routines (similar to a monitor as defined by Hoare [7]) which interrogates a system-wide table (in a shared segment) of currently active files to determine if concurrent usage of a particular file is allowable at any given point.[8]

---

[8] See [8] for more details on the structure of the file system in MTS as well as details on how concurrent usage of files is accomplished in MTS.

Protection of Information in Files

For the protection of information in files, most state-of-the-art time-sharing systems use access control lists associated with each file, and MTS is similar in this respect. Briefly, when a file is initially created in MTS, the owner of the file has unlimited access to perform any of a number of operations on the file. These operations might include, for example, reading the file, writing (with a distinction made between expanding and changing), emptying (discarding the contents), renumbering, truncating (deallocating unused disk space), renaming, destroying (deallocating all disk space), and permitting (giving access to others). Initially everyone else has, by default, no access to the file. Subsequently, the owner of a file may give specific users of the system permission to perform any combination of the above-mentioned operations on the file.

Each user of MTS is identified by a unique identification code, (hereafter referred to as a userid), and as is the case with most systems, a password mechanism is used to authenticate a particular user at SIGNON time. Once signed on, that user has access to all of his own files as well as to other users' files to which his userid has been given explicit access. In addition, users at the University of Michigan are grouped into projects -- for example, all students in a particular class or all staff on a particular research project -- and the owners may permit their files to be accessed in a specified fashion by all users within a particular project (i.e., all users with a specific project number).

Control of Access to Files

Two things should be mentioned about the file-sharing facility. First, every user of the system has at least one set of access rights associated with any given file (in most cases it is, by default, no access to the file). Second, if a user has more than one set of access rights to the file because both his userid and his project have been given permission to access the file, then a priority scheme is used to determine the access rights. The set of access rights associated with the userid is used if given -- if not, then the set of access rights associated with the project is used if given. Otherwise, the access rights associated with everyone else is used (by default no access, but changeable by the owner). Thus, in the usual case, the owner of a file specifies that certain specific users have certain specific kinds of access to a file, while everyone else has no access to it. However, the owner may also specify that everyone has some specified access, except for certain users or groups of users who have some other access, e.g. none.

Although the mechanisms described so far are seemingly quite general and flexible, they have at least one major shortcoming. Specifically, if a user has been given "read" access to a file, he might read that file in a number of

different ways, each obtaining quite different amounts of information. For example, if the file contains a program (i.e., an object module) to be executed, the user might simply request that MTS read the file for the purpose of loading and executing the program. Alternatively, the user might request that a system utility be invoked which will read the file and tell him about the internal structure of the object module. Or he might read the file directly himself and make a copy for his own use. Each usage requires only that the user has been given read access to the file, but each obtains successively more information about the contents of the file. Likewise, permission to allow a user to write a file can have varying consequences depending on how the user chooses to exercise his "write" privileges. In the least destructive case he may only be appending information to a file or to each record in the file; in the most destructive case he may be completely changing or deleting every record in the file. It thus seems that a more discriminating access control mechanism is needed.

Program-Controlled Access to Files

Saltzer and Schroeder [6] refer to the need for "protected subsystems", i.e., user-provided programs which control access to files. As they indicate, only a few of the most advanced system designs have tried to provide for user-specified protected subsystems and these have, in general, been with special hardware (or extensive software) designed to assist in the implementation. Honeywell's MULTICS [5] uses the hardware ring structure to provide protected subsystems to control access to files, whereas more current (experimental) systems such as the CAP system of Cambridge University [9] and the HYDRA system of Carnegie-Mellon University [10] use hardware-or-software-implemented capabilities, respectively, to provide such protected subsystems.

MTS addresses this need by providing a software-implemented access control list mechanism for allowing user-written protected subsystems. That is to say, by extending the permission mechanism to allow data files to be accessed in specified ways by programs as well as by users and projects, MTS allows a specific user-written program -- i.e., a file which contains an object module to be executed -- to be designated the only thing allowed particular types of access to specific files (which might contain data, for example). Thus the designer of a program can, if he desires, completely control the read and write access to his data files. Specifically, this mechanism is provided by allowing the owner of a file (containing an object module to be executed) to associate a unique identification (an eight-character string called a "program key") with that file. Then data files, for example, may be specified as accessible in a particular way (read, write change, destroy, etc.) only by a particular program key -- i.e., only if the executing program which is accessing the data file was loaded from a file with the appropriate program key. Program keys are prefixed internally

by a userid, and thus they are unique among users. In addition, they are in general non-transferable. In this way users can be restricted from having any direct access to data files; only an executing program may then access the data files, presumably in the fashion prescribed by the designer of the program.

Execute-Only Files

Obviously, the users of the program (with program-key access to the data files) must also have been given some sort of access to the file containing the program in order to run it. This could simply be read access as described before, but a software implemented "execute-only" access is provided as a specific example of the general concept of program-key access to files. (As mentioned previously, the machines on which MTS runs have no hardware implemented execute-only storage access type; thus the motivation to implement something in the software.) Execute-only access is accomplished in MTS by associating a unique program key with each of the system programs invoked when a user enters a command, in particular the system program which loads and executes user programs. Thus, permission to read files containing programs to be executed (i.e., object modules) may be given only to the system load-and-execute program (i.e., the RUN command interpreter). In a similar manner, files may be designated as "edit-only." For example, only the system context editor program might be given permission to write a file containing the source for some program, thereby guaranteeing that the source file will not be accidentally damaged by a careless user or even by the owner.

Priority Resolution of Multiple Access Rights

Quite obviously, the addition of program keys to the list of those who may access a file complicates the priority resolution when the userid, project, and program-key combination have more than one set of access rights to a file. In MTS, the priority of program-key access to files is lower than userid or project access, but higher than the global access associated with everyone else. Thus, if both the userid and/or the project which initiated execution of the program, as well as the program key of the program accessing the file, have been given explicit access to the data file, the access rights associated with the userid (or then the project) will be used to determine the type of access allowed. Only if the userid and project which initiated execution of the program have no explicit access to the data file is the access associated with the program key used. Of course, if the program key of the currently executing program has also not been given explicit access to the data file, then the global access associated with everyone else is used. It should be noted that all files (in particular those which contain object modules) are given a default program key when they are created. Thereafter, the owner of the file may change the program key to any "legal" value he desires (i.e., in

general one prefixed with his userid).[9]

Applications

One of the most obvious applications where it is desirable to have a particular program in complete control of access to data files is that of a data base management system. In such a situation in MTS, the users could be given execute-only access to the data base management program itself, and that program could be given read (and possibly write) access to the data base files. At this point it becomes clear (or at least it became clear during the implementation), that one would like to be even more explicit in specifying what users, projects, and programs may access a file. In particular, one would like to be able to specify that only particular users and/or projects have specific types of access to data files, but only if they access the files by running specific programs. In fact, the facilities provided in MTS are general enough, for example, to permit only a specific project (e.g., a student class) to have read access to the data base files for inquiry purposes, but only via the data base management program; and at the same time to permit other specific users (e.g., the instructors) to have both read and write access to the data base files for updating purposes, but still only via the data base management program.[10]

Implementation Considerations

Since in MTS the protection of information in files is completely implemented in the software, it might be useful to discuss some of the more interesting implementation considerations. In particular, MTS goes to considerable lengths to guarantee that a program with an explicitly assigned program key (or an execute-only program) is run in a manner consistent

---

[9] As an indication of the amount of sharing of files which actually goes on in MTS at the University of Michigan, the following numbers may be of interest. One should keep in mind that the facility for sharing files among users and projects has been available for close to 4 years, whereas the facility for giving programs access to files has been available for only about one year. Currently in MTS there are over 86,000 private files on-line. Of these, over 40,000 are shared in some way; the others are accessible only to their owners. Over 22,000 are shared by everyone, and over 27,000 are shared by specific users, projects, or programs. Obviously, some of the 40,000 are shared both globally and specifically. Of the 27,000 files accessible specifically to users, projects and programs, over 1,000 have been given program-key access.

[10] See [11] for details on exactly how this is accomplished in MTS, and how the now even more complicated priority of multiple access rights is resolved.

with the desires of a security-conscious program designer. In general, this means that if MTS detects such a program being run as anything other than a single, self-contained "load module", MTS will insure that program key access to the data files is denied.[11]

Concerning other areas of implementation strategy, Saltzer and Schroeder [6] refer to the complications involved when one considers the "dynamics of use" of protection mechanisms. In particular, they refer to how one establishes and changes the specifications of who may access what files. In MTS, the owner of a file always has authority to change the access control list. That is to say, there is no way that the access to a file can be forever denied to everyone. In addition, anyone to whom the owner has given "permit" access may also alter the access control list of the file. Obviously, the owner must be careful in giving away the right to change the access control list. Since changing the access control lists dynamically may imply potential problems with access rights to currently active files no longer being valid, MTS determines the access that a userid, project, and program key combination has to a file the first time the file is referenced (opened), and retains that information in a control block in the system work segment. MTS also maintains global information (in a shared segment) to indicate how and by whom each active file in the system is currently being accessed. Thus, when someone attempts to change the access control list, MTS is able to determine whether the file is currently active (and as a consequence whether there are access rights outstanding in a control block in storage) and will not allow the access control list to be changed until all activity associated with the file has ceased.

User-Controlled Domain Switching

Linden [4] mentions another area of interest when implementing protection mechanisms, that of providing a mechanism for changing from one protection domain to another with potentially different access rights to files as control passes from one protected subsystem to another. As mentioned previously, programs executing in a process will regularly switch (under system control) between three protection domains, and the access to the segments in the process's virtual storage will change accordingly. Similarly, a user at the command language level may switch between a number of different command language subsystems during a typical terminal session, and the access to files may change accordingly. The most commonly used subsystem is the main command interpreter, but other subsystems include, for example, the editor subsystem, the program debugging subsystem, and the user program execution subsystem.

---

[11] See [11] for details on how this is accomplished.

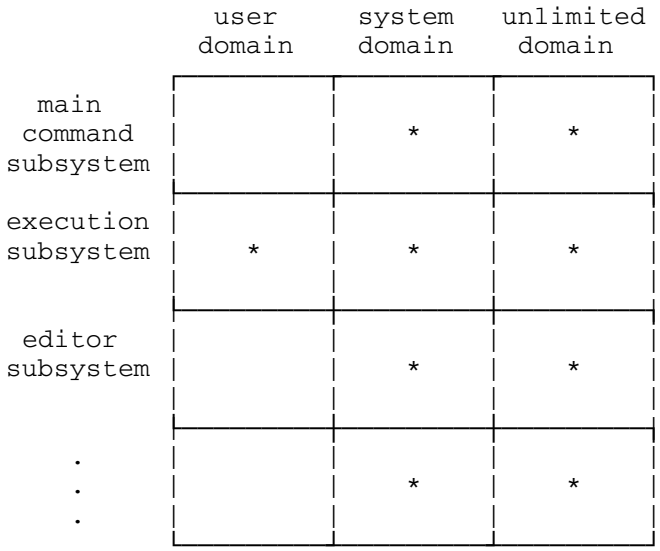
It should be noted that before program-key access to files was implemented, the access which a userid and project had to a particular file was invariant during the time the file was active (open). This is no longer true when one considers the access which a userid, project, and program key combination has to a file. The access to the same file may potentially be different depending on which subsystem is currently accessing the file. For example, if the context editor has been given read and write-change access to a file, and also a particular program (via its associated program key) has been given write-expand access to the file, then when the user is editing the file he may read and change lines in the file (but not add lines). If he switches to the program execution subsystem to run the above-mentioned program, he may add lines to the file (but not read or change them). Needless to say, MTS must be aware when control switches from one subsystem to another and reevaluate the access rights, if necessary.

Actually, as indicated in figure 3, the various protection domains in MTS intersect. For example, a program running in the execution subsystem may switch between the user and system domains and as a result the access to segments will change accordingly. However, if the program remains in the execution subsystem, the access to any currently active files will be invariant. On the other hand, a user switching between the main command subsystem and editor subsystem may have different access to a file currently in use, but if both subsystems remain in the system domain the access to segments will be invariant. Finally it should be noted that the execution subsystem is the only subsystem which runs in the user domain.

One important facility which the program key mechanism is not able to provide and which could not be easily provided with software alone, is the ability to have multiple protection domains (with different program-key access to files) within an executing program -- for example, upon entry to a user callable subroutine. It seems clear that such a facility requires additional hardware and/or extensive software assistance, probably in the form of the above-mentioned capabilities to allow a feasible solution.

Summary

By controlling access to information in virtual storage, MTS provides an environment in which processes are protected from one another, and the operating system is protected from individual processes. The system control data structures used by MTS are either in the shared segments or in the system work segment of each process, and are thus protected from errant or malicious user programs. In particular, the data necessary for the control of access to files is out of the reach of user programs. In addition, by allowing users a flexible mechanism for controlling how their on-line files may be accessed and by what users, projects, and programs, MTS provides a general



Intersection of protection domains in MTS.

Figure 3

facility for the sharing of information in files in a (program controlled) manner specified completely by the owner.

While some of these mechanisms are not as general as others which have been proposed -- particularly capability-based mechanisms -- they do provide a flexible, secure environment while at the same time maintaining system integrity. More generality in the mechanism would require either modifications to the 360/370 architecture or a great deal more overhead in the software.

References

[1] D. W. Boettner and M. T. Alexander. "The Michigan Terminal System." pp. 912-918, Proceedings of the IEEE, Special Issue on Interactive Computer Systems, Vol. 63, No. 6, (June 1975).

[2] IBM System/370 Principles of Operation. IBM Publication GA22-7000-4.

[3] B. Arden, B. Galler, T. C. O'Brien, and F. Westervelt. "Program and Addressing Structure in a Time-Sharing Environment." pp. 1-16. Journal of the ACM, Vol. 13, No. 1, (Jan. 1966).

[4] T. A. Linden. "Operating System Structures to Support Security and Reliable Software." pp. 409-445. ACM Computing Surveys, Vol. 8, No. 4, (Dec. 1976).

[5] J. H. Saltzer. "Protection and the Control of Information Sharing in MULTICS." pp. 388-402, Communications of the ACM, Vol. 17, No. 7, (July 1974).

[6] J. H. Saltzer and M. D. Schroeder. "The Protection of Information in Computer Systems." pp. 1278-1308, Proceedings of the IEEE, Vol. 63 No. 9, (Sept. 1975).

[7] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept." pp. 549-557, Communications of the ACM, Vol. 17, No. 10, (Oct. 1974).

[8] G. C. Pirkola. "A File System for a General Purpose Time-Sharing Environment." pp. 918-924, Proceedings of the IEEE, Special Issue on Interactive Computer Systems, Vol. 63, No. 6, (June 1975).

[9] R. M. Needham and R. D. H. Walker. "Protection and Process Management in the CAP Computer." pp. 155-160, Proceedings of the IRIA International Workshop on Protection in Operating Systems. Institut de Recherche d'Informatique et d'Automatique, France, 1974.

[10] E. Cohen and D. Jefferson. "Protection in the HYDRA Operating System." pp. 141-160, Proceedings of the Fifth ACM Symposium on Operating Systems Principles, ACM Operating System Review, Vol. 9, No. 5, (Nov. 1975).

[11] MTS Manual, Vol. 1: The Michigan Terminal System, "Files and Devices, Appendix I," pp. 153-160. The University of Michigan Computing Center, Ann Arbor, Michigan, April 1976.