TENEX Protection Memo

Chuck Wall
June 14, 1974

Technical Memo

TM.74-20

1. Introduction

This memo is a first draft of some ideas on how to provide a more general protection mechanism for TENEX users. We will review the relevent access control models and implementations, consider some applications that the present TENEX protection mechanisms do not support in general, specify the basic approach to be used in providing new protection mechanisms, analyze the TENEX system with respect to the protection models and additional requirements and provide some proposed solutions. All of these ideas are at best proposals at this stage. It is felt that it is appropriate at this time to expose these ideas to both the BBN TENEX group and the I4 group in order to provoke criticism which will provide us with a better framework and guidelines for proceeding.

There will be some detailed implementation specification memos that go along with this memo and extend the basic ideas covered here.

2. Review of relevent access control models and implementations

We consider "access control mechanisms" or "protection mechanisms" to be those mechanisms (hardware, firmware or software) "which control the access of a program to other things in the system." [L71] It is usually the case that we treat a process (a program and its operating environment) as subjects and the "other things" as objects. [e.g. G68, J73, L69a, L69b, L71, WU73]

A general model proposed in various forms by Graham and Denning, Lampson, Jones et. al. is the access control model which specifies the following three components:

Objects - those entities to which access must be controlled;
Subject - an active entity whose access to objects must be controlled;
Access - the operation or the access a subject can potentially perform on an object.

Jones [J73] describes access protection in terms of enforcement at each of three sites: the "object site," the "execution site" and the "access site." The three basic problems to be solved in a specific implementation of this model are as follows:

1. Identify the subject and object types and specify the set of possible accesses to each object type;

2. Specify representations for the protection information associated with each object type;
3. Insure the enforcement of the protection rules;
4. Specify the mechanisms and the rules under which subjects can manipulate protection information.

We will assume that each object can be uniquely referenced, that subjects and accesses can be considered object types in order that they may also be protected and that objects are composed of constituent components that may require access protection.

Two general conceptual representations of protection information that allow the specification of alternative implementation representations are the access matrix and triple. (see fig. 2.1) The access matrix is a three dimensional matrix representation where the various subject, object and access types are associated with one dimension of the access matrix. Clearly, not all subjects have access to all objects nor do all access types apply to all object types. Indeed it is expected that an access matrix would be quite sparse and not directly implementable since considerable space would wasted. Hence, it is important to consider alternative encodings. An encoding that would be equivalent to the access matrix is the triple (Sub, Obj, Access). We could implement a dynamically changing list of such triples. This would conserve space but would be costly in terms of the time involved to search the list and validate the access. It is convienent then to

Access Matrix:

SUBJECTS: FORKS	Forks			JFNs		Directories		Files	
	FK1	FK2	FK3	J1	J2	D1	D2	F1	F2
FK1		Halt Kill		Byte-Input		Login Logout		Read	
FK2									
FK3			Block Wakeup		Random I/O		Connect		Read Write

Triple:

(Sub, Obj, Access)

Alternative encodings:

Matrix partition	Tuple	Enforcement at	Typical Use
Row	(Obj, Access)	Execution site	Capability
Col	(Sub, Access)	Object site	Access control
Plane	(Sub, Obj)	Access site	Type checking

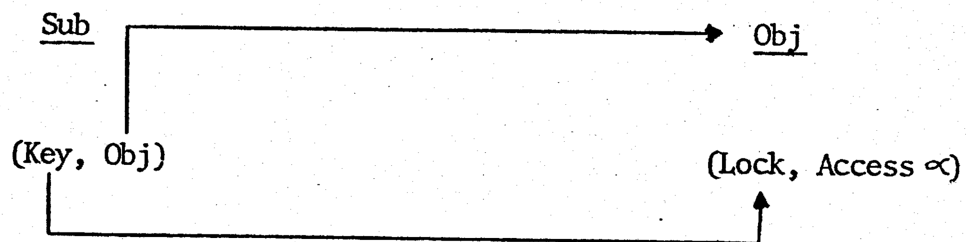
Fig. 2.1: Representation of Protection Information

consider partitioning the access matrix. If we want to control the abilities that a subject has we could specify an encoding of (Obj, Access) pairs in the subject's environment commonly refered to as a "capability list." In order to control access to objects that have an existence independent of subjects we could encode the (Sub, Access) pair in the object's environment. This type of encoding is usually refered to as an "access control list" and is associated with file and directory types of object. Finally, we can associate protection with the access the pair (Sub, Obj). A common use of this encoding is type checking. For example, in the prologue for the code to do a particular access it is usual to check that the object is of the appropriate type.

Another popular mechanism that provides an extension to the encodings already presented is the key and lock mechanism. [G68] (see fig. 2.2) In this encoding a subject has access to a (Key, Obj) pair which is used to attempt some access, β to the object specified in the pair. The object has access to a (Lock, Access α) pair which is used to verify the access. When some access β is attempted to an object, the access is allowed if the key presented is equal to a lock protecting that object and access β is a subset of access α. Note that we do not have to explicitly know who the subject is in this encoding, it is sufficient that the subject has presented the correct key and access type.

Key and Lock Mechanism:

To allow sets of subjects access to sets of objects



When a subject attempts Access β to an Object

If Key = Lock & $\beta \leq \alpha$ Allow access

Else fail.

Note:

Internal Keys and Locks must be unforgeable. A subject must have access to a key by name only.

Fig. 2.2: Key and Lock Representation

The key and lock mechanism is useful in a number of ways. For example:

- When a user is trying to login to a system there is usually something like a "universal subject" with limited powers that is acting on behalf of this as yet unidentified user. In this case the user would typically supply a "password" as a key and a "user name" in conjunction with the "login" access. If the "password" key supplied by the user matches the "password" in the lock location of the object associated with the "user name" and the access associated with that lock is "login" then the login is allowed and the user is associated with a unique subject;
- When a set of subjects, each requiring the same access, must be associated with an object;
- When a subject must have access to a set of objects.

Internally coded keys and locks must be unforgable. The subject must not be able to manufacture keys and locks, but must request the system to supply them and give the subject access by name only. In the "password" example used above the "password" that is supplied by the user is an external key. In general it is not a wise idea to allow programs to employ external keys for access control.

3. Some applications that presently pose protection problems

Many of the applications problems posed here are from the I4-TENEX installation. Some of the problems have been combined and reformulated into a more general category. These problem areas will be useful in determining which objects in TENEX will require more immediate attention. Each problem will now be described in an individual paragraph.

There is a need to allow particular subsystems specified access to particular files. It is possible that the access could include "delete". Currently subsystems that need this facility are SNDMSG, SPOOLER, DUMPER and a SNOBOL program that is used to get mail for a group of users.

When a subsystem is running in an inferior fork of a job can we provide a protected environment? That is, can we provide protection up to and including the level that is currently associated with jobs? This question implies that it will be possible for a subsystem to exercise abilities that the job under which it is running and all other parallel or superior forks do not and should not have. (See Fig. 3.1)

Particularly at I4, there is a resource allocation assignment difficulty and I quote Dr. Pirtle on this: "Frequently a job with minimal CPU allocation will need to use a service which needs a large CPU allocation in order to provide the service promptly. Generally, the service uses a non-pre-emptable gadget (the I4 or the B6700)."

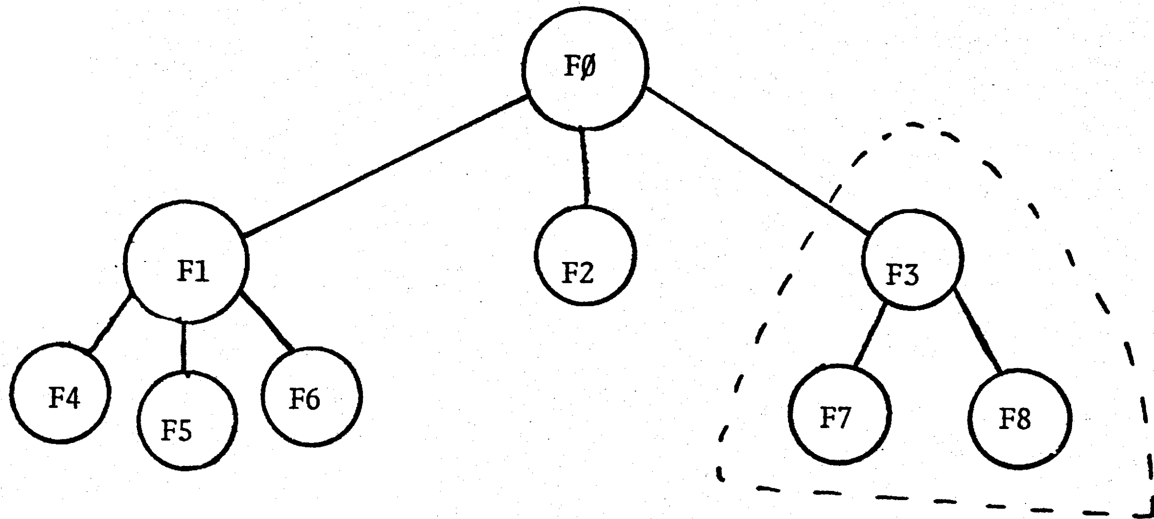
This problem appears to me at least to be a difficult one in terms of the current TENEX implementation and in fact is currently being handled by I4 code extensions. Dr. Pirtle has proposed three alternative approaches to the solution of this problem:

- Create a fork that requests access to the gadget and a large CPU allocation. When the access is granted provide the I4 service routine the required CPU allocation
- Have the service routine run in an independent job to which the user passes access to files, connections to terminals, etc.
- Provide a way in which each process can identify itself, the "owner" of the job of which it is a part, and other identifying information to a centralized resource manager. This manager will respond to requests for resource allocations.

When a user wants to submit an I4-Job he wants the I4-Job executed by a TENEX job which has access to the minimum number of files necessary to do the job. Also, this TENEX job must have its own CPU allocation, but the user's disk allocation.

We would like to be able to set up an environment for subsystems such that they cannot be misused or have their code analyzed. It is possible now (in fact, DDT is always merged into the virtual address space of the last subsystem fired up under the EXEC) to merge in code that can either directly examine the subsystems code or if it is protected in execute only pages to systematically execute every instruction, analyze the results and attempt to construct the algorithm or possibly the code involved.

Some Job:



Can F3, F7, F8 be given job like qualities:

- Isolated from other forks
- Be connected to different directories
- Have different abilities and restrictions

Fig. 3.1: Fork protection for subsystems

4. Basic approach to the implementation of new protection mechanisms

The basic approach to be followed will be to identify the subject, object and access types. Evaluate each object type in terms of the protection model and the problem areas in order to determine the general protection mechanisms to be used. Determine the level(s) of protection to be specified. The detailed specifications for each object type to be considered will be covered in separate memos. These memos will specify the protection mechanism implementation, the rules under which subjects can manipulate protection information and enforcement of the rules.

5. Analysis of TENEX with respect to protection

In this section we will attempt to identify all the relevant subject and object types in TENEX. In so doing, we will identify several basic object types to be protected. Each of these object types will be composed of constituent components. For example, a JOB is an object which by itself must be protected. A FORK is a constituent component of a JOB which itself must be protected from illegal access. We could of course consider forks as objects and specify a mechanism that would protect FORK type objects. Since forks of one job are presently independent forks in another job in TENEX we will consider the notation JOB.FORK to stand for the name of an object type, different from object type JOB. To carry this a little further consider a PAGE as a constituent component of either a FILE or a JOB.FORK, then FILE.PAGE and JOB.FORK.PAGE would be considered different object types.

We note also that our determination of object types and access types is an arbitrary one at best. If for instance we have the following:

Object type:	Access
JOB.FORK:	Kill Freeze . . . Read fork AC's Set fork AC's Read fork PC Set for PC . . .

could just as easily be specified as:

```
Object type:        Access

JOB.FORK:           Kill
                    Freeze
                    .
                    .
                    .
JOB.FORK.STATE:     Read fork AC's
                    Set fork AC's
                    Read fork PC
                    Set fork PC
```

or as

```
Object type:        Access

JOB.FORK:           Kill
                    Freeze
                    .
                    .
                    .
JOB.FORK.AC:        Read
                    Set
                    .
                    .
                    .
JOB.FORK.PC:        Read
                    Set
```

or for that matter, a large number of alternative ways. The main point of this is that while our selection of object and access types will be arbitrary in the sense of what is possible, we will attempt to justify it in terms of the current implementation and the problems that it presently presents.

The basic objective in adding new protection mechanisms to TENEX is to provide for more protection on objects and to provide a more flexible environment with respect to the problems being experienced at I4. In particular we want to allow subsystems to be protected as objects when they are put into a fork, to be identified and to be able to specify capabilities

different from and perhaps "amplified" beyond the job's capabilities. To allow this we need the following services and abilities:

- A method of specifying "protected subsystems". (See memo on Protected Programs)
- A set of two JSYSes that allow a protected GET (PGET) and a protected SAVE (PSAVE).
- The ability for a protected subsystem to establish its environment, protect itself from superior access (see memo on fork protection), associate itself with a directory that the job does not have access to (see memo on directory numbers) and bring with it from that directory DDBPRV word capabilities that are different from and perhaps "amplify" the capabilities the job has in general.

Since the only subject in TENEX is the JOB.FORK and since we will be allowing "protected subsystem" to become subjects and to protect themselves from their superiors, it is desirable to consider some extensions that would allow the superior fork to be able to protect itself from the subsystem. To this end we recommend the following:

- Extend the capabilities word by adding bit assignments in bits B9-B17 as follows:

  B1Ø - Allow fork to initiate PSI on superior channel;
  B11 - Allow fork to get superior trap word; and
  B12 - Allow fork to read status of superior.

- Allow the superior fork to specify the mask for bits B9-B17.

The basic objects to be considered for protection in TENEX are as follows:

DEVICE
DIRECTORY
DIRECTORY'NUMBER
FILE
JOB
JOB.ACCOUNT
JOB.JFN
JOB.FORK
JSYS
SYSTEM

We feel that the current protection on DEVICE, JOB, JOB.ACCOUNT, JSYS and SYSTEM object types is sufficient at this time and will not consider them in this memo. We note however that BBN has considered a form of protection on JSYS object types. We also point out that CFORK and LOGIN can be considered as operations on the SYSTEM since they create new objects and subjects.

The protection mechanisms to be implemented concern the remaining objects:

DIRECTORY
DIRECTORY'NUMBER
FILE
JOB.JFN
JOB.FORK

The access control list and the implementation is treated in detail in a set of separate memos.

6. How do the proposed mechanisms solve the problems

Subsystems that need specific access to particular need only be put on the access list for that file or be given an internal key. Subsystems such as SPOOLER, DUMPER, SNDMSG and the get mail SNOBOL program should not contain "password" keys.

Subsystems can be protected as we have specified in the fork protection memo.

The resource allocation problem will be handled by a central resource manager.

Forks that need a CPU allocation could be allowed to set JOBBIT in their PSB. Right now only forks that contain protected subsystems would have the ability to do this. We have not completely specified an implementation to handle this for other forks.

Environments for subsystems will be set up by PGET.

BIBLIOGRAPHY

[GD72] Graham, G.S. and Denning, P.J., "Protection - Principles and Practice," AFIPS 1972 SJCC Proceedings, Vol. 40, AFIPS Press, Montvale, N.J., 417-429.

[J73] Jones, Anita K., Protection in Programmed Systems, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1973.

[L69a] Lampson, B.W., CAL-TSS Internals Manual, Computer Center, University of California at Berkeley, November 1969.

[L69b] Lampson, B.W., "Dynamic Protection Structures," AFIPS 1969 FJCC Proceedings, Vol. 35, AFIPS Press, Montvale, N.J., 27-38.

[L71] Lampson, B.W., "Protection," Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems, Department of Electrical Engineering, Princeton University, Princeton, N.J., March 1971, 437-443.

[WU73] Wulf, W.A. et al, "HYDRA: The Kernel of a Multiprocessor Operating System," Carnegie-Mellon University, Computer Science Department report, June 1973.