ESD-TR-75-57 MTR-2931

DESIGN OF A SECURE FILE MANAGEMENT SYSTEM

J. C. C. White

APRIL 1975

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

ADA 010590

REVIEW AND APPROVAL

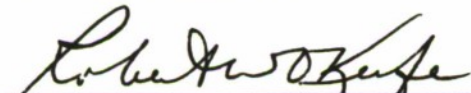This technical report has been reviewed and is approved for publication.

WILLIAM R. PRICE, 1Lt, USAF
Techniques Engineering Division

ROGER R. SCHELL, Major, USAF
Techniques Engineering Division

FOR THE COMMANDER

ROBERT W. O'KEEFE, Colonel, USAF
Director, Information Systems Technology
Deputy for Command & Management Systems

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-75-57	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DESIGN OF A SECURE FILE MANAGEMENT SYSTEM		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER MTR-2931
7. AUTHOR(s) J. C. C. White		8. CONTRACT OR GRANT NUMBER(s) F19628-73-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation Box 208 Bedford, MA, 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 7070
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems Electronic Systems Division, AFSC Hanscom Air Force Base, Bedford, MA, 01731		12. REPORT DATE APRIL 1975
		13. NUMBER OF PAGES 30
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) COMPUTER SECURITY FILE SYSTEM PRIVACY SECURITY		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A file management/operating system based on the PDP-11/45 Security Kernel is described. The system will allow complete sharing of files, subject to the control of the Security Kernel, so that problems brought about by the conflicting requirements for security and sharing can be identified and explored. It will provide a vehicle for experimentation with the extensions of the kernel required for multisource information correlation.

LIST OF ILLUSTRATIONS

Page

2

SECTION I

INTRODUCTION

A security kernel based file management/operating system is presently under development at MITRE. It will be used as a demonstration and test vehicle for the PDP-11/45 security kernel[1], and will provide a suitable environment for experimentation with the extensions of the kernel required for operation on such problem areas as multisource information correlation. Since the system is being constructed as part of a fairly modest effort, for experimental rather than general, widespread use, it provides simple, limited capabilities. It is, however, sufficiently flexible to allow meaningful exploitation of the kernel's capabilities. In particular, it allows complete sharing of files, at all security levels, subject to the control of the security kernel, so that the problems brought about by the conflicting requirements for security and sharing can be identified and explored. It also provides full machine-language programming and debugging capability, including direct (not interpretive) control of many peripheral devices (those which do not do direct memory access without CPU intervention). In those instances where a choice between simplicity and speed had to be made, speed was considered to be of secondary importance; the principal objective was to produce a suitable and reliable system.

The system is being coded in the Project SUE System Language, and provides convenient support for subsystems and user programs written in that language. These programs are first processed by the System/370 SUE compiler and link editor, and may then be loaded into the PDP-11/45 via tape, at system start-up time, or by punched card thereafter. Machine-language programs from any other source may also be introduced into the system via punched cards or even from the users' terminals, providing complete flexibility for any penetration/information compromise experiments which may be attempted.

This document describes the fundamental design and user interface of the secure file management system. Since the system is still in the development stage, this description should not be considered definitive; it is intended instead to identify an approach, and to clarify some of the problems encountered in building a system based on the security kernel approach. Familiarity with the security kernel concept on the part of the reader will be assumed, in order to avoid the tiresome repetition of notions that have been described elsewhere[1,2]. However, specific details of the PDP-11/45 security

3

kernel will occasionally be identified and described, where such details are particularly critical to the design of the secure file management system.

The file system described here closely resembles that of the MUMPS "global array",[4] in its logical structure. Because of the constraints imposed by the kernel, its physical structure is somewhat more complex.

SECTION II

THE FILE SYSTEM

FILE STRUCTURE

The principal structural element of the file system is referred
to as a block. A block contains a collection of items, which are
individual data elements; these elements are identified by 16-bit
subscripts, which are unique within a given block. The blocks com-
prising the file system are arranged in a tree-like hierarchy, so
that the entire structure is ultimately descendant from a single
Root Block. Access control is on a block basis, so that a process
which can operate on a given item within a block can operate, in
the same mode, on any other item in the same block.

The contents of an individual item are a data element and/or a
downward pointer. The data element, as presently defined, is either
fixed-length numeric or variable-length string. Other data types
might be added, if their inclusion would provide a useful extension
to the system. The fixed-length numeric type, chosen to correspond
to the PDP-11 word, is 16 bits long. The first 8-bit byte of the
variable length string determines the string's length L ($1 \leq L \leq 72$),
and is followed by the L bytes of the string. The downward pointer
points to a block which is immediately descendant from the current
block, thereby defining the block hierarchy already mentioned.
Alternatively, the downward pointer may point to a single data segment
with no system-defined structure. The data segment is the fixed-size,
non-directory unit of (virtual) memory controlled and allocated by
the security kernel. Such segments are used to store executable
procedures, and may be used for any other information for which the
block storage is not suitable. An implementation-dependent maximum
of 48 downward pointers may emanate from a single block. Figure 1
provides an example of a section of the file structure.

A given block within the file system is identified by a sequence
of subscripts, naming items containing downward pointers, starting
from the Root Block. It may be observed that a more or less Multics-
like file system could be built with this basic structure, using
blocks as directories and data segments as Multics segments (albeit
rather abbreviated ones). However, in general, blocks may be used
to store any data, not just pointer-related information, so that
most subtrees of the file system will not contain any unstructured
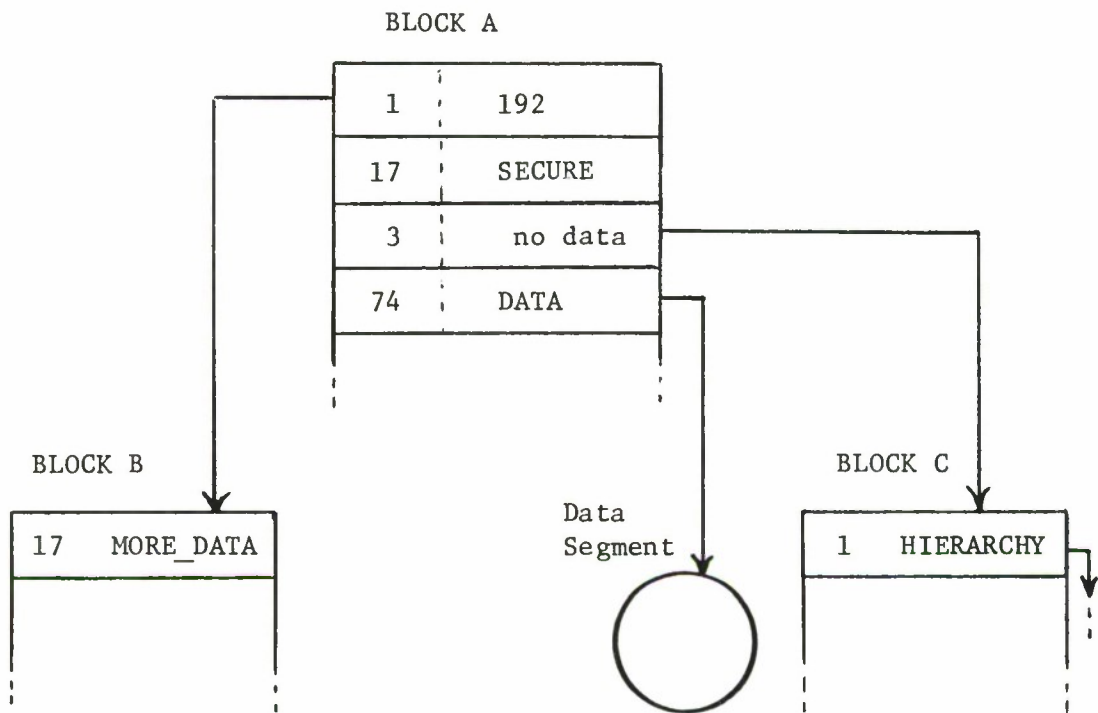data segments.

Figure 1. File Structure

6

BLOCK STRUCTURE

The blocks must be constructed from segments, the basic storage objects allocated and controlled by the security kernel. An individual block is made up of a directory and from one to fifteen data segments, as shown in Figure 2. A data segment is always present at offset position 1, below the directory, and other data segments are added, in positions 2 through 15, as required. Each data segment of a block contains a 16-bit word-count, which defines the number of words occupied by items in that segment, and a continuation pointer, which points to the next segment making up the block (actually, it is the offset of that segment in the block's directory, or zero if no more segments are included in the block). In addition, the first data segment of each block contains a 16-bit Indicator, which is used to facilitate sharing of data under the constraints imposed by multi-level security. The Indicator's use will be described in the section on file sharing. The remainder of each data segment is composed of items, compacted so that all unused space in a segment is contiguous, at the upper end.

Individual items have four parts:

a) a 16-bit subscript, uniquely identifying the item within its block. Subscript 0 and $FFFF_{16}$ are not allowed, as they have special significance to the file system; any other subscript is legal;

b) an 8-bit set of flags, indicating the type and significance of parts (c) and (d);

c) an 8-bit pointer which, if significant, specifies an offset in the directory segment included in this block. The segment identified by this offset will be either (i) the directory segment associated with a lower-level block, or (ii), an unstructured data segment;

d) the data, either a fixed-length numeric or a variable-length string, as described earlier. Alternatively, this part of the item may be empty (and occupy no space) if the flags so indicate. An item must include either a significant pointer or significant data, or both.

In order to read or write an item with a specified subscript in a given block, the block must be linearly searched for that subscript. The length of each item may be determined by examination of one or two bytes, so that the search may rapidly proceed from one

7

FROM
HIGHER–LEVEL
BLOCK

DIRECTORY SEGMENT

DATA
SEGMENT 1

| Word–Count |
| Continuation Pointer |
| Indicator |
| Data Item 1 |
| Data Item 2 |

| Data Item n |
| Unused Space |

DATA
SEGMENT 2

| Word–Count |
| Continuation Pointer |
| Data Item n + 1 |

TO
LOWER–LEVEL
BLOCK
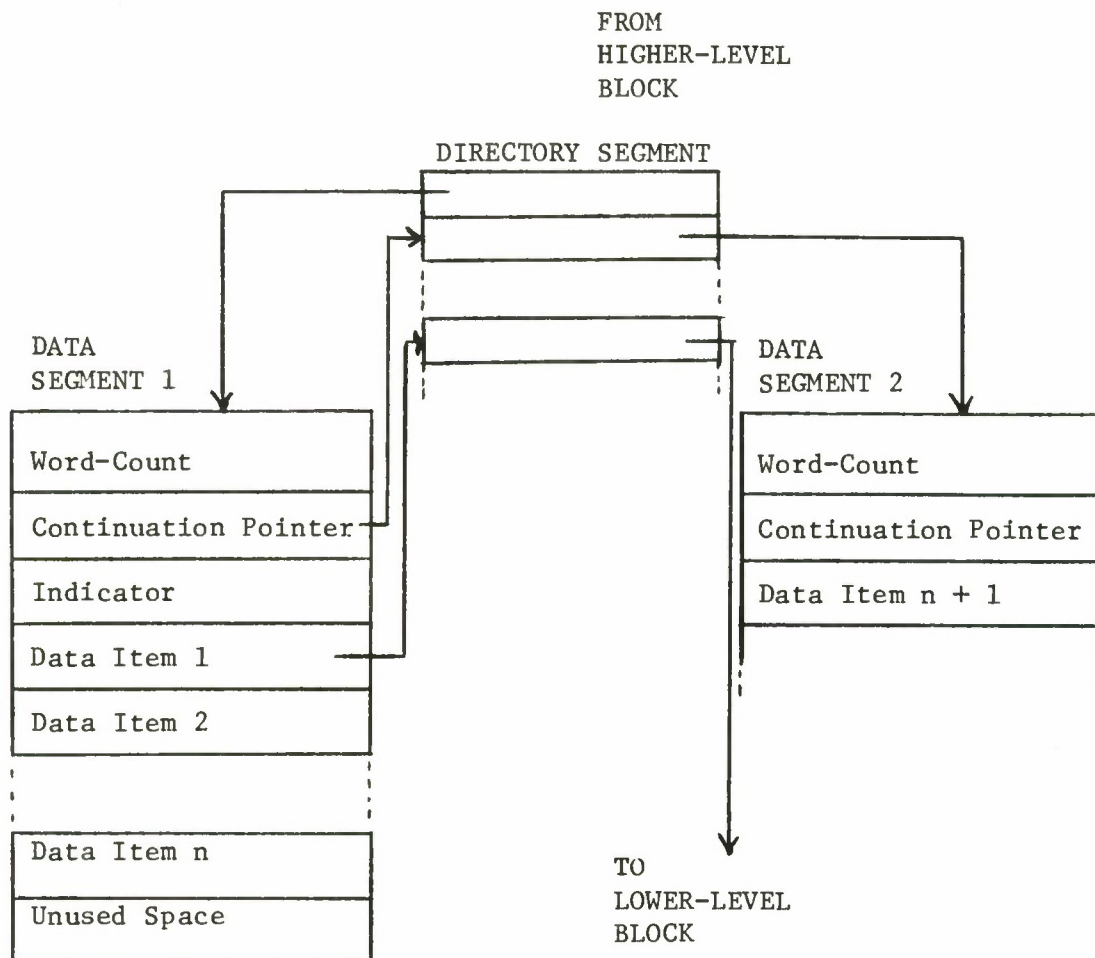
Figure 2.   Block Structure

8

WORD 1

SUBSCRIPT	
FLAGS	POINTER

WORD 1   SUBSCRIPT

2   FLAGS | POINTER

(a)   Item with pointer only

1   SUBSCRIPT

2   FLAGS | 0

3   NUMBER
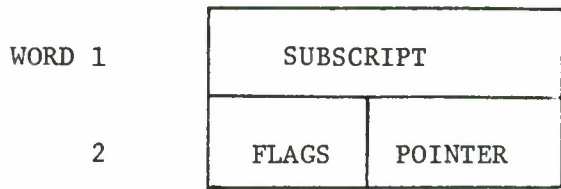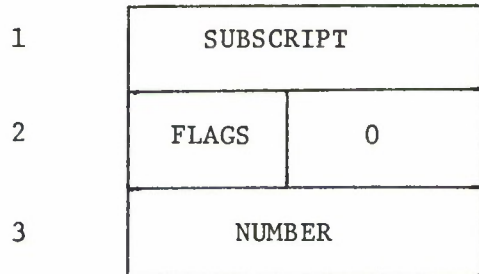
(b)   Item with numeric data only

SUBSCRIPT

FLAGS | POINTER

L | DATA

REQUIRING

L BYTES

FOR STORAGE
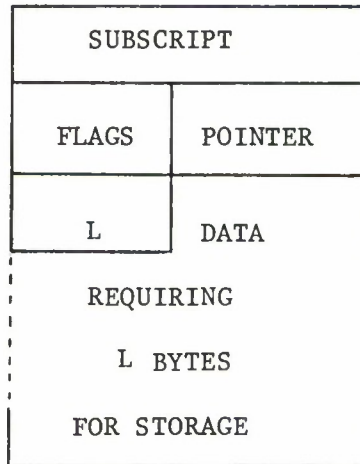
(c)   Item with pointer and string data

Figure 3.   Item Structure

9

subscript to the next. Nevertheless, this method makes the fifteen-segment limitation on block size quite appropriate; the maximum block size allowed is 15k bytes, or as much as 60k bytes if the 4k byte segment size is implemented in the kernel. Larger files, not structured by the file system, can be constructed if necessary with the use of pointers to individual data segments.

BLOCK ACCESS CONTROL

The control of access to information in the secure file management system occurs at the block level. The command which creates a block specifies its classification and category, which apply to both the directory and the data segments which comprise a block. Since the search for a subscript is linear, this is the finest level of security control allowable on blocks. However, the security attributes of unstructured data segments are defined at the individual segment level.

The granularity chosen for access control has a substantial influence upon the file organization appropriate for a given application. The most attractive level of access control, from the user's point of view, would probably be the ability to specify access attributes of individual items or small collections of items, as well as those of entire files. The burden of maintaining separate segments (the basic objects of the security kernel) for the items of various security levels in a given file would then be placed upon the file management system. As a consequence, the file management system would require data definition language, a great deal of structure, and all the attendant machinery to support them. The alternative chosen for this system provides the user with a very simple and flexible structure, but requires him to arrange his files so that their blocks correspond to the desired security organization.

One other option available to the application programs is the use of a trusted process, implementing a certified sanitizing algorithm, to move information from storage of a given security level to that of a lower level. This method would allow the storage of all information in a file at a single, high, security level, and the subsequent extraction of lower levels of data via the trusted process. Complex, data- and environment-dependent access control could thereby be achieved. The mechanisms required involve significant overhead, and must be certified correct, which limits their use to situations where careful design of files is inadequate. The approach is extensively discussed elsewhere[3].

10

The principal access regulation enforced by the security kernel
involves the comparison of security levels of processes and segments.
A secondary mechanism, the Access Control List (ACL), is also pro-
vided. Since the information maintained in the ACL is provided by
user-mode programs via the file management system, it must be under-
stood that it does not provide the same sort of absolute protection
that the primary security level control does. Specifically, the ACL
mechanism in the security kernel is certified to be correct, but the
policy it enforces is under the control of uncertified programs
(the file management system and user programs).

A potentially awkward aspect of the file system's tree structure
lies in the fact that a process with proper classification, properly
authorized on a block's ACL, may still be unable to access that
block. The anomaly arises when the process does not have at least
read access to every block in the chain leading from the root to the
desired block. The security kernel will detect an attempted access
violation, and refuse to allow further access, when the first such
block in the chain is encountered. In the MULTICS system, which has
a similar file hierarchy, the problem is circumvented by allowing all
processes sufficient access to step through the chain, regardless of
ACL information. If this approach were taken here, it would require
modification to the security kernel, with the particularly unattrac-
tive effect of specializing the kernel for the secure file manage-
ment system (since the ability to step through a chain of blocks
implies some form of read access to both directory and data seg-
ments).

The approach chosen for this system is to verify that a (poten-
tial) user already has access to every block in the chain at the
time access is given to a particular block. If not, the user will
still be added to the block's ACL, but an error indication will
alert the calling program to the existence of the inconsistency.
That program may ignore the error, or correct its cause, as required.

BLOCK OPERATIONS

The nature of the operations available on the file system may
now be briefly examined. At any instant in time, one block in the
file system (per process) is specified to be the current block (at
process startup, the Root Block is the current one). Operations are
performed on the current block, and a command is available to change
the block specified as current to any block in the system which is
available to the requesting process. The new current block may be

11

specified by its position relative to the old current block, or relative to the Root Block.

The operations involved in reading an item are elementary. The desired block is established as current. When the subscript identifying the item to be read is specified, the block is searched until either the subscript is found, or the block is exhausted. In the former case, the data contained in the block is returned; in the latter, the return code signals failure.

Writing an item is somewhat more involved, but still straightforward. First, the desired block must be made the current one. Next, that block must be searched for an existing item with the specified subscript. If one is found, its pointer, if significant, is saved, the existing item is expunged, and the segment is compacted. The new item may then be written in the first available space in the block, which may require the appending of a new segment to the block. The flowchart of Figure 4 should clarify the process.

Finally, the operations required to change the current block will be outlined. One parameter of the CHANGE-BLOCK command specifies a subscript-list, which contains the sequence of subscripts to be followed in locating the new block. An initial subscript of zero specifies that the sequence is relative to the Root Block; initial subscripts of $FFFF_{16}$ move the starting point for the rest of the sequence one block up the tree, toward the Root; any other subscripts are relative to the current block. The sequence is terminated by a non-initial zero subscript. The file system maintains a list of the blocks leading from the root to the current block, so that the upward operation can be performed without benefit of backward pointers.

FILE SYSTEM PROCEDURES

The procedures listed below may be called by user-level programs. The procedure name and parameters are specified for each, along with a brief description of its effects.

WRITE_STRING (SUBSCRIPT, DATA)

Alters the current block so that the command READ_STRING (SUBSCRIPT), with the same block current, will return the string DATA.

12

Figure 4. WRITE_STRING Procedure (Parameters are subscript and new item to be written)

13

READ_STRING (SUBSCRIPT)

>   Returns the string associated with SUBSCRIPT in the current
>   block; if no such string exists, returns 0 as the string length.

WRITE_NUMERIC (SUBSCRIPT, DATA)

>   Alters the current block so that the command READ_NUMERIC
>   (SUBSCRIPT), with the same block current, will return the word
>   DATA.

READ_NUMERIC (SUBSCRIPT)

>   Returns the numeric data word associated with SUBSCRIPT in the
>   current block; if no such data word is stored, returns the
>   word 0.

TYPE (SUbSCRIPT)

>   Returns the TYPE, which may be STRING, NUMERIC, or NULL, of the
>   data associated with SUBSCRIPT in the current block.

DELETE_DATA (SUBSCRIPT)

>   Deletes the data associated with SUBSCRIPT, if any, in the
>   current block. Subsequent READ operations to the current
>   block, with that subscript, will return the word 0 (unless a
>   WRITE with that subscript and block has intervened).

CREATE_BLOCK (SUBSCRIPT, CLASS, CATEGORY)

>   Creates a new block, and a pointer to that block, associated
>   with SUBSCRIPT, in the current block. The new block is
>   initially empty, has security level specified by CLASS and
>   CATEGORY, and has no access allowed (GIVE_ACCESS must be used
>   to make the new block accessible). If a pointer is already
>   associated with SUBSCRIPT, if a compatibility violation is
>   attempted, or if the process cannot write the current block,
>   returns FALSE; otherwise returns TRUE.

DELETE_BLOCK (SUBSCRIPT)

>   Deletes the pointer associated with SUBSCRIPT in the current
>   block, the block it points to, and all other blocks subordinate
>   to that one. Returns FALSE if the deletion is illegal (process
>   cannot write the current block), TRUE otherwise.

14

CHANGE_BLOCK (SUBSCRIPT_LIST, MODE)

Makes the new current block the one identified by the sequence of subscripts in SUBSCRIPT_LIST. If the initial subscript is 0, the sequence starts at the root; otherwise, at the old current block. A subscript of 177777_8 refers to the parent of the block reached by the preceding sequence; all others refer to the child block identified by the subscript, in the block reached by the preceding sequence. MODE is READ or WRITE. Returns FALSE if the block specified cannot be reached or made current in the mode specified; otherwise, returns TRUE.

CURRENT_ID

Returns a subscript-list giving the name relative to the root of the current block.

NEXT_SUBSCRIPT (SUBSCRIPT)

Returns the value of the next higher subscript than SUBSCRIPT which exists in the current block, or 0 if none exists.

GIVE_ACCESS (SUBSCRIPT, MODE, USER, PROJECT)

First, verifies that the USER-PROJECT combination has at least READ access to the current block, and to every block above it in the hierarchy. Then if the current block can be written, gives MODE-type access, to the block or segment pointed to by SUBSCRIPT from the current block, for the USER-PROJECT combination. Returns TRUE if the access is given and already exists for all blocks higher in the chain, FALSE otherwise. In this way, only meaningful access can be given without an error signal, since inability to read a given block implies that all blocks subordinate to that one are inaccessible.

RESCIND_ACCESS (SUBSCRIPT, USER, PROJECT)

Rescinds access to the block or segment pointed to by SUBSCRIPT from the current block, for the USER-PROJECT combination. Does not verify that no "orphan" access rights, to descendant blocks, are thereby created.

CREATE_SEGMENT (SUBSCRIPT, CLASS, CATEGORY, SIZE)

Creates a new data segment of the specified SIZE, identified by the pointer associated with SUBSCRIPT in the current block. The new segment has security level specified by CLASS and

15

CATEGORY. Returns FALSE if a pointer is already associated
with subscript, if the current block cannot be written, or
if a compatibility violation would occur; TRUE otherwise.

GET_SEGMENT (SUBSCRIPT, SAR, MODE)

Brings the data segment pointed to by SUBSCRIPT from the current
block into the (virtual) memory accessed via Page Address
Register SAR, and enables it for operations of the specified
MODE. If SAR is already in use, the segment associated with it
is first dismissed. Returns FALSE if the segment is inaccessible
or nonexistent, true otherwise. Confusion (but no security
compromise) is likely to occur if SAR specifies the memory
presently pointed to by the PC or the stack pointer.

DISMISS_SEGMENT (SAR, FUTURE-USE)

The specified SAR is disabled. If FUTURE USE is FALSE, the
segment will be released; if TRUE, it will remain activated
unless the process segment Descriptor Table overflows, at which
time it may be released.

HALT

Returns control to command level, in supervisor mode.

FILE ORGANIZATION

An example of a simple organization of information in the file
system is shown in Figure 5. This arrangement is by no means the
only one possible for this system, but it appears to adequately
serve the intended applications.

The Root Block, created at initialization and directly sub-
ordinate to the kernel root, points to the basic subtrees of the
file system. The first of these subtrees is the Subsystem Library,
which contains the principal collection of programs to be run on the
system. The topmost block of this subtree consists of items con-
taining the names of the various subsystems; in the example, these
are EDITOR, CARD_INPUT, etc. Each of these items points to a block
which contains all information required to load and run the sub-
system itself. In particular, this block contains pointers to the
code segments which make up the subsystem. It may also contain any
other information necessary for the initialization or operation of
the subsystem, as determined by its own specific requirements.

16

ROOT BLOCK

1	LIBRARY
2	USER_FILES
3	SYSTEM_ DIRECTORY_ BLOCK
4	DATA_BASE_1
5	DATA_BASE_2

<proj1>
<proj2>
<proj3>

1773	EDITOR
	CARD_ INPUT

1
2
3

<user1>
<user2>

<user1>
<user5>

1	PROCESS_1_ TEMPORARIES
2	PROCESS_2_ TEMPORARIES

STACK

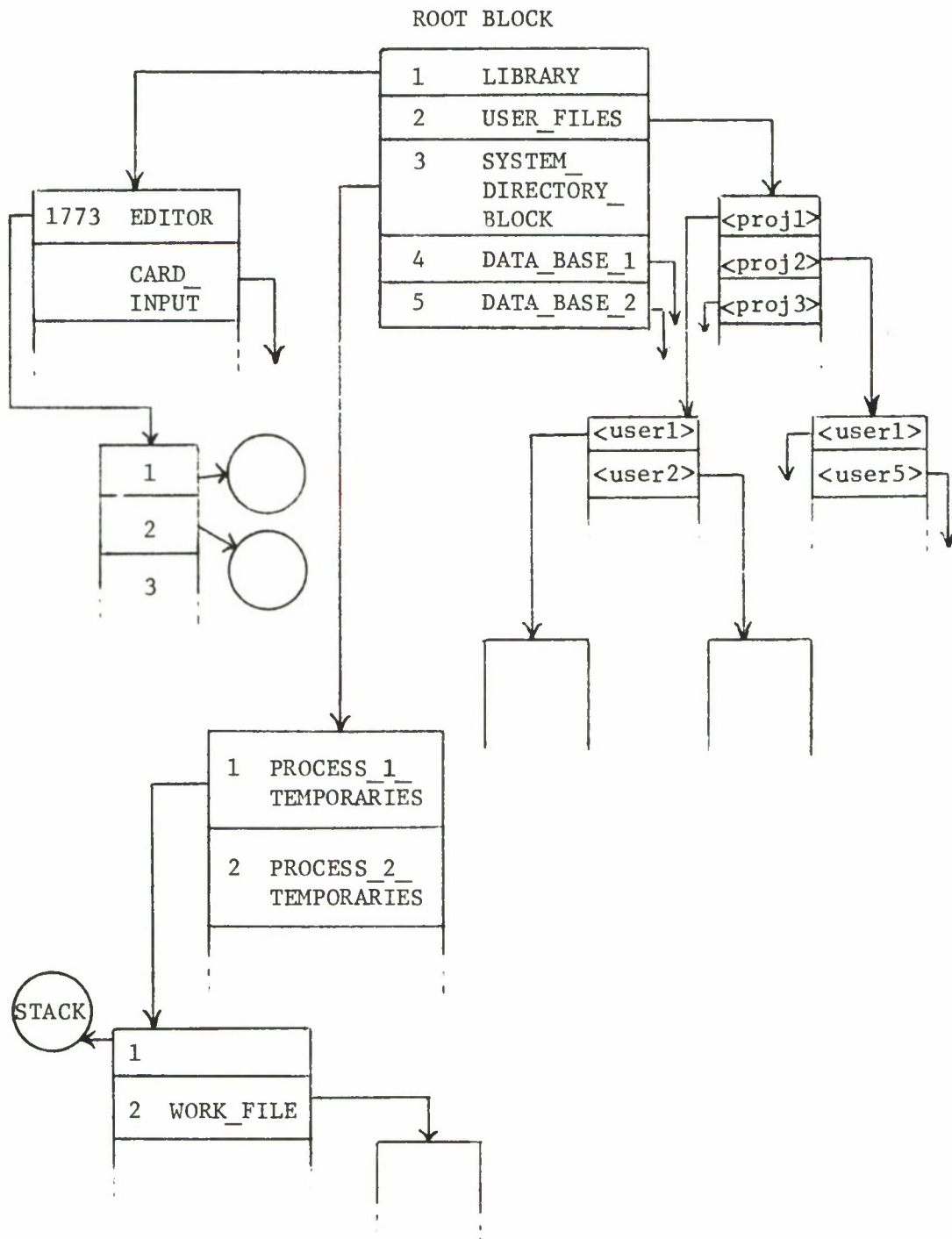| 1	
2	WORK_FILE

Figure 5. Sample File Organization

17

The second subtree of the file system is the collection of User Files. These provide permanent storage for files peculiar to individual users and projects. The first block of the User Files subtree consists of items with subscripts corresponding to the one-word names of the various projects authorized to use the system. Each of these items includes a pointer to a subordinate block whose items correspond to individual users associated with that project. These items may then serve as base pointers to whatever files are to be associated with a particular user-project combination. Whenever a new user-project pair is authorized access to the system, a new base pointer for the exclusive use of a subject identified by that pair may be created.

The third item of the Root Block provides a pointer to the System Directory Block. This block contains an item for each process number which can exist on the system (i.e., 1-15 for the current kernel implementation). These items point to the individual Process Directory Blocks, which are created when a process is established, and have classification, category, and ACL so that the corresponding process (and only that process) has read/write access to it. This block is deleted when a process terminates, so that it is a convenient location for storage of temporary data, including user stack segments.

The remaining items of the Root Block are available for pointers to shared data bases. Since these are highly dependent upon individual applications, their characteristics will not be amplified until such applications are defined.

SECTION III

SUBSYSTEM LOADER-DEBUGGER

After a user has logged onto the system (via a certified "answering service" program), he is placed in contact with the subsystem loader-debugger routine, which operates in supervisor mode. That routine's principal function is to allow the loading and running of user-mode programs. In addition, it provides some debugging aids: breakpoints, memory and register read and write, and execution of a limited number of user instructions followed by a return to command level (i.e., to contact with the loader-debugger). This last capability seems particularly important, since allowing the user program control of its I/O devices leaves no way of halting a looping program from the user's terminal. Operation in this mode is much slower, however, so that it is intended only for debugging purposes.

Subsystems' initial segments are normally stored as data segments pointed to by blocks subordinate in the file structure to block 0, 1. (Recall that 0 specifies the root block). The subscript identifying a particular subsystem is simply the word formed by the first two characters of the subsystem name, so that these two characters must be unique for each one. For example, the EDITOR is identified by ED (ASCII) = C5C4 (Hexadecimal) = 50628 (decimal). Then a pointer to the subsystem block of the EDITOR is stored at subscript 50628 of block 0, 1. In addition, the string EDITOR is stored as the data associated with that subscript, as a check. The following pairs of characters are reserved for debugging commands, and may not be used as the initial characters of subsystem names: GE, LO, GO, BP, QU, Rn, and nm, where n and m are any decimal digits.

LOADER-DEBUGGER COMMANDS

<string>

This is the basic subsystem loader command, where <string> does not have one of the reserved pairs as its initial characters. The initial segment of subsystem <string>, (located at subscript 1 of its subsystem block), is located as described above, and made accessible, in READ mode, through User SAR 6 (i.e., it becomes segment 6 of the user's virtual memory). An empty stack segment is established, under this process's block of temporaries, addressable through SAR 0. Registers R0-R5 are set to zero, R6 to the second highest word of the stack segment, and R7 to the lowest word of the (code) segment

19

addressed by SAR 6 (these assignments conform to the conventions of SUE System Language programs). Control is then passed to the user mode, starting at the instruction indicated by R7. It is of course assumed that the specified segment is executable code; unintended results may otherwise ensue.

LOAD < subscript-sequence > $\lvert$ W $\rvert$

The segment identified by subscript-sequence is loaded, a stack is established, and R0-R7 are initialized as described above, but execution is not started. Optional parameter W specifies that the named segment is to be enabled in WRITE mode. < subscript-sequence > is a series of decimal integers $\leq (2^{16}-1)$, separated by commas. Examination and alteration of memory and register contents, and setting of breakpoints, may be done before executing the GO command.

GO $\lvert$ n $\rvert$

Starts operation, in user mode, at the location specified by R7. Optional parameter n specifies, in decimal, the number of instructions to be performed before returning to command level. This control is implemented by use of the Trace Trap feature of the PDP-11/45, so that its use causes the user program to run very slowly.

QUIT

Dismisses all user segments; deletes temporary segments, including the user stack; executes the kernel command STOPP.

BPON < memory location >

Sets a breakpoint at the specified location in user virtual memory (which must be enabled in WRITE mode). < memory location > is a hexadecimal integer $\leq FFFF_{16}$. An attempt to execute an instruction at this location causes BREAK to be printed, and control to be returned to command level.

BPOFF $\lvert$ < memory location > $\rvert$

Removes the breakpoint at the specified location, if any. BPOFF alone removes all breakpoints (a maximum of eight may be set at one time). Any breakpoint in a segment is removed automatically when that segment is dismissed from the SAR.

BPLIST

Lists the memory locations at which breakpoints are currently set.

<memory location> $\left|,n\right|$

Types the contents, in hexadecimal, of n consecutive memory locations starting with the specified one. The default value of n is one.

Rn

Types the contents, in hexadecimal, of user register n, where $0 \leq n \leq 7$.

<memory location>   =   <word>

Changes the contents of <memory location> to <word>, which is a hexadecimal integer $\leq$FFFF. The location must be enabled in WRITE mode.

Rn   =   <word>

Changes the contents of register n to word.

SAMPLE SUBSYSTEM: EDITOR

The EDITOR serves here as an example of a subsystem, which runs in user mode and operates with the assistance of the file system. It is a simple, interactive, text editor, which is capable of manipulating string-type information within a block. It is called by typing EDITOR from command level, or, alternatively, by typing

LOAD   0,1,50628,1

GO

as explained in the previous sections. A brief description of the editor commands follows. Each command consists of a one-or two-character command code, followed by parameters, separated by spaces unless otherwise specified. They all operate on the current block, which is established by the first command. Note that the editor indicates readiness for a new command line by printing *. All commands operate with the help of a Current Pointer (CP), which in general indicates the most recently affected line.

21

* F p_1, p_2,..,pn	<u>Fetch</u> the block identified by the sequence of parameters, and establish it as the current block. Set the CP to 0. If no such block exists, create it (if possible), with the current process classification and category, and give this process access to the block.
* I p_1 p_2	<u>Insert</u> lines, starting at subscript p_1, incrementing line numbers by p_2. Lines are typed in, delimited by carriage returns. Stop when a line consisting of just a period (.) is typed. Default value of p_2 is 1. Default value of p_1 is CP + 1.
* P p_1 p_2	<u>Print</u> lines, starting at the first subscript $\geq p_1$, in increasing order of subscripts, stopping when no more subscripts $\leq p_2$ can be found. Default value of $p_2 = p_1$. Default value of $p_1 = $ CP + 1. Set CP to the subscript of the last line printed.
* PN p_1 p_2	Same as <u>P</u>, but subscripts are <u>not</u> printed by this command.
* D p_1 p_2	<u>Delete</u> all lines with subscripts s so that $p_1 \leq s \leq p_2$. Default value of $p_2 = p_1$. Default value of $p_1 = $ CP. Subscript of last line deleted entered into CP.
* L string	<u>Locate</u> and print the first line containing <string>, starting at line CP. Set CP to the subscript of that line.
* R/<string 1>/<string 2>/p_1	<u>Replace</u> p_1 occurrences of<string 1> with <string 2>, starting at line CP. Default value of $p_1 = 1$. Set CP to the subscript of the last line altered. Print the new lines.
* X	Return to command level

SECTION IV

SECURITY KERNEL REQUIREMENTS

FILE SHARING

The file system described here is intended to be shared by
several users concurrently. It is therefore necessary to provide
some sort of control to prevent simultaneous accesses by more than
one process from producing anomalous results. This control is pro-
vided by a set of semaphores, one for each segment currently access-
ible by any process on the system. Any process requiring exclusive
access to a segment might issue a P to the appropriate semaphore;
upon completing its work, a V would make the segment available to
the next process requiring such access. However, such a simple
solution conflicts with the security restrictions imposed by the
kernel. Consider, for example, the case of an unclassified segment.
The semaphore associated with this segment must be unclassified,
since it may be manipulated by uncleared processes. However, the
*-property then prevents a classified process from using this sema-
phore, so that the classified process cannot protect the item from
the possibility of being altered by some unclassified process while
it is being read. Since a process must be capable of reading items
of lower security level, this problem cannot be ignored.

The security kernel enforces the restriction that a segment may
be written only by processes of a single, specific, security level.
The semaphore may therefore be used to prevent two processes from
attempting to write to the same segment simultaneously. Since seg-
ments will, in general, be read by processes of higher security
level, the best that can be done when reading is to detect situations
in which an item may have been altered while it was being fetched,
so that another fetch may be attempted. For this purpose a one word
Indicator is included in the first data segment of each block. The
Indicator must be incremented by any process which writes in that
block. Then any process which reads the block may compare its
Indicator, before and after reading, to determine whether the block
has been changed during that time period. This can be done with no
modification to the kernel. However, it is still necessary to pre-
vent a process from initiating a read of a block, while a write to
the same block is in progress. For this purpose, a binary-valued
semaphore test procedure, called T, will be used. T (i) has value
True if semaphore i is negative, False otherwise. The sequences

23

for reading and writing a block whose first data segment is segment
i are then as follows:

Read

CYCLE

    CYCLE

        a:=Indicator (i);
        EXIT UNLESS T(i);
    END;
    .
    .
    .

    Read block here
    .
    .
    .

    EXIT WHEN a = Indicator (i)

END

Write

P (i)

Indicator (i) := Indicator (i) + 1
.
.
.

Write block here
.
.
.
V(i)

The procedure T is purely a read; it is bound by the security rules
for reading (where the semaphore has the security attributes of its
associated segment). It may be desirable to have T(i) cause the
current process to relinquish control of the processor when True
for the sake of efficiency.

To see that these sequences perform the intended function, it is necessary to recognize that the $T(i)$ test can only be passed when no process is writing the block under consideration. Then any alteration of that block which is initiated before the $T(i)$ test must be completed before the read occurs, and cannot lead to a change during the read; and an alteration which begins after the $T(i)$ test will first cause the Indicator to change, so that the read will be repeated. It is clear that the read may be delayed for an indefinite period by a succession of writes, but the rules of the security kernel appear to make this inevitable, whatever the technique chosen.

The use of the mechanisms described here is implicit in the file system commands. They will protect the user from conflict-induced inconsistencies, without any need for awareness on his part. It may prove desirable to allow explicit use of the semaphores by user-level programs, but no such capability is presently specified. It will be necessary, in the latter case, to make some provision for cleaning up any semaphores erroneously left set by a terminated process.

OTHER REQUIREMENTS

The following are other initial and operating conditions which should be provided in order for the system to operate as described here.

1. Process Directory Block. The executive process should create such a block, of appropriate classification/clearance, at the time a new process is started. The process itself cannot create the block, since it requires writing into the (unclassified) System Directory Block. The stack segment supplied by the startup procedure should be placed subordinate to the Process Directory Block.

2. Trap Handling. Calls from one mode to another will be made by the various trap instructions of the PDP-11. Calls to the kernel are made with the TRAP instruction. The kernel should accept such calls only when they are made from supervisor mode, so that the file system's organization may be protected, to some degree, from untoward happenings in user-mode programs. Kernel calls from user mode should produce no action other than an immediate return from the kernel.

In order to facilitate user-supervisor mode communications, the EMT and BPT trap vectors, located in kernel address space, should

direct control to supervisor mode, specific locations to be determined. Supervisor mode handling of these traps will allow programming of breakpoints and fetching of parameters for file system calls to be performed with no unnecessary demands made upon the kernel.

3. Cleanup. The Process Directory Block, and any subordinate segments or blocks, should be deleted, by the executive process, when the STOPP command has been issued. It might also reset any semaphores which have been left set by the STOPP-ing process, though there should be none in the case of a normal exit. This step would be highly desirable in the case of abnormal termination of a process.

SECTION V

CONCLUSION

The basic design of a secure file management system has been presented here. The system is intended to provide a useful interface between the minimal structure provided by the security kernel and the generalized, unpredictable requirements of user programs. Some details of the design have been left unspecified, since they may better be filled in as experience is gained with the kernel and with the SUE language. However, there are two areas which require further detailed design, since their proper operation is essential to the intended function of the system.

First, a downgrading mechanism must be integrated with the file system design. The downgrading mechanism's basic characteristics have been described[3], but the implementation and procedures to be used have not been specified. Since the system's use in multisource information correlation depends upon its ability to move information from one level to another in a safe, certified manner, this capability must be added to the system.

Second, a mechanism for communication between procedure segments is lacking. Since the kernel presently supports only 1k byte segments, the virtual memory available to the user is discontinuous, and some convenient means of bridging the discontinuities is required. An alternative approach would be to modify the kernel to support 8k byte segments, so that the discontinuities could be eliminated.

The design of the secure file management system has provided assurance that the PDP-11/45 security kernel is powerful and flexible enough to form the basis for a useful system. The particular problems of file identification and sharing have been treated in some detail, and the approaches described here are believed to be generally applicable to security kernel based systems. Further details will be documented as the design evolution continues.

27

REFERENCES

1. Schiller, W. L., "Design of a Security Kernel for the PDP-11/45", ESD-TR-73-294, December 1973.

2. Bell, D. E., and LaPadula, L. J., "Secure Computer Systems", ESD-TR-73-278, Vol. 1-3, 1973, 1974.

3. Stork, D. F., "Downgrading in a Secure Multilevel Computer System: The Formulary Concept", ESD-TR-75-62, May 1975.

4. Greenes, R. A., et al, "A System for Clinical Data Management", Proceedings of the 1969 Fall Joint Computer Conference, pp 297-305.