ESD ACCESSION LIST
DRI Call No. 88496
Copy No. 1 of 2 cys.
FILE COPY

ESD-TR-77-259, Vol. II

MTR-3294, Vol. II

DESIGN AND ABSTRACT SPECIFICATION OF A MULTICS SECURITY KERNEL

BY P. T. WITHINGTON

MARCH 1978

Prepared for

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

Approved for public release;
distribution unlimited.

Project No. 522N
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-77-C-0001

ADA 053148

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

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

Sylvia R. Mayer

SYLVIA R. MAYER
Acting Chief
Techniques Engineering Division

William R. Price

WILLIAM R. PRICE, Captain, USAF
Techniques Engineering Division

FOR THE COMMANDER

Stanley F. Dereska

STANLEY F. DERESKA, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Command & Management Systems

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ESD-TR-77-259, Vol. II
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE *(and Subtitle)*: DESIGN AND ABSTRACT SPECIFICATION OF A MULTICS SECURITY KERNEL

5. TYPE OF REPORT & PERIOD COVERED:

6. PERFORMING ORG. REPORT NUMBER: MTR-3294, Vol. II

7. AUTHOR(s): P. T. Withington

8. CONTRACT OR GRANT NUMBER(s): F19628-77-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. 522N

11. CONTROLLING OFFICE NAME AND ADDRESS:
Deputy for Command and Management Systems
Electronic Systems Division, AFSC
Hanscom Air Force Base, MA 01731

12. REPORT DATE: MARCH 1978

13. NUMBER OF PAGES: 114

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
FORMAL SOFTWARE SPECIFICATION
MULTICS
SECURITY KERNEL

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*:
On the basis of the recommendations of the Electronic Systems Division Computer Security Technology Panel (1972), The MITRE Corporation developed techniques for the design, implementation, and formal mathematical verification of a security kernel: a hardware and software mechanism to control access to information within a computer system.

(over)

20. Abstract (continued)

This three-volume report describes the design of a security kernel for Honeywell Information System's Multics computer system. This second volume gives a formal, top-level specification of the primary subsystems of the kernel. The specification is a definition of the input-output behavior of the kernel. It is sufficiently detailed to allow its security, compatibility, and efficiency to be determined.

The first volume gave a methodology and design overview. The third volume deals with the secondary subsystems, including the issues of initialization and reconfiguration.

ACKNOWLEDGMENT

This report has been prepared by The MITRE Corporation under Project No. 522N. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

This report describes a design that has been evolving since September 1974. A number of individuals have contributed, including W.L. Schiller, who was responsible for the major part of the storage management design; S.R. Ames, Jr., K.J. Biba, E.L. Burke, M. Gasser, S.B. Lipner, and J.P.L. Woodward of The MITRE Corporation; and Lt. Col. R.R. Schell, Capt. W.R. Price, and Capt. P.A. Karger of the U.S. Air Force. The design has also been influenced by discussions with personnel from other Air Force contractors and subcontractors: Honeywell Information Systems, the Computer Systems Research group of M.I.T's Laboratory for Computer Science, and Stanford Research Institute's Computer Science group.

TABLE OF CONTENTS

		Page
LIST OF ILLUSTRATIONS		6
LIST OF TABLES		7
SECTION I	INTRODUCTION	8
	BACKGROUND	8
	SYNOPSIS	8
SECTION II	SPECIFICATION	10
	FORMAL LANGUAGE	10
	SYNTAX AND SEMANTICS	11
	Basic Definitions	12
	The Functions	13
	Semantic Conventions	14
	ISOLATION OF THE KERNEL	15
	MEDIATION MECHANISMS	15
	THE MODULES	16
	Common Basic Definitions	16
	Common Functions	21
	Common Module--Conclusion	24
	SPECIFICATION REVIEW	24
SECTION III	THE INTERPRETER CONCEPT	25
	TECHNIQUE	25
	APPLICATION OF THE INTERPRETER	26
	INTERPRETER SUPPORT IN THE KERNEL	27
	COMPATIBILITY WITH THE CURRENT MULTICS	27

TABLE OF CONTENTS (continued)

		Page
	SPECIFICATION	28
	Basic Definitions	28
	V-functions	28
	O-functions	28
	VERIFICATION WITH THE INTERPRETER	33
	INTERPRETER REVIEW	35
SECTION IV	STORAGE SYSTEM	36
	CURRENT DESIGN--DETAILS	36
	THE KERNEL DESIGN	38
	Access Control	38
	Finiteness	38
	Correctness	39
	COMPATIBILITY WITH THE CURRENT MULTICS	40
	Directory Images	41
	The Quota Mechanism	43
	Message Segments	44
	Backup and Retrieval	44
	SPECIFICATION	44
	Unique Identifiers	44
	Basic Definitions	45
	Hidden V-functions	47
	V-function Macros	50
	V-functions	57
	O-functions	57
	STORAGE CONTROL REVIEW	76
SECTION V	PROCESS MANAGEMENT	78
	CURRENT DESIGN--DETAILS	78
	Process Creation and Deletion	78
	Interprocess Communication	79

TABLE OF CONTENTS (concluded)

		Page
	THE KERNEL DESIGN	79
	COMPATIBILITY WITH THE CURRENT MULTICS	80
	SPECIFICATION	81
	Basic Definitions	81
	Hidden V-functions	83
	O-functions	85
	PROCESS CONTROL REVIEW	90
SECTION VI	EXTERNAL INPUT/OUTPUT	91
	CURRENT DESIGN--BACKGROUND	91
	THE KERNEL DESIGN	93
	Communications I/O	93
	Peripheral I/O	94
	Input/Output Coordinator Support	96
	Operational Requirements	96
	COMPATIBILITY WITH THE CURRENT MULTICS	97
	SPECIFICATION	97
	Basic Definitions	97
	V-functions and O-functions	99
	EXTERNAL INPUT/OUTPUT REVIEW	102
SECTION VII	SUMMARY	106
APPENDIX I	INDEX TO SPECIFICATIONS	107
APPENDIX II	SPECIFICATION LANGUAGE SYNTAX	109
REFERENCES		112

LIST OF ILLUSTRATIONS

Figure Number		Page
1	Common Type Definitions	17
2	Common Parameters	19
3	Common Constants and Definitions	20
4	Access Lattice Functions	22
5	Time and Audit Functions	23
6	Interpreter Module Basic Definitions	29
7	Read and Execute Functions	30
8	Write and Test_and_set Functions	31
9	Implications of the Interpreter	34
10	Quota Cell Mechanism	37
11	Hierarchy with Directory Images	41
12	Storage Control Type Definitions	46
13	Storage Control Parameters and Constants	48
14	Storage Control Primitive V-functions	49
15	Inas Function	51
16	Access_permission and Acle_apply Functions	52
17	Quota Functions	54
18	Implementation Functions	55
19	Storage Verification Functions	56
20	Seg_attributes and Seg_side_effect_attributes Functions	58
21	Update_quota Function	59
22	Mount and Demount Functions	61
23	Initiate, Terminate, and Revoke_access Functions	62
24	Create_segment Function	65
25	Delete_segment Function	68
26	Add_ACL_element and Remove_ACL_element Functions	70
27	Enter_msg Function	73
28	Move_quota and Release_page Functions	74
29	Process Control Type Definitions	82
30	Process Control Parameters, Constants, and Hidden V-functions	84
31	Create_proc and Delete_proc Functions	86
32	Block, Interrogate, and Wakeup Functions	88
33	Send_wakeup and Set_principal_id Functions	89
34	Kernel I/O Structure	92
35	Communications I/O Paths	94
36	Peripheral I/O Paths	95
37	External I/O Basic Definitions	98
38	Send Function	100
39	IOM Functions	101
40	Copy_segment Function	103

LIST OF TABLES

Table Number		Page
I	Kernel Segment Attributes	40
II	Directory Image Attributes	42
III	Storage Control Access Modes	45

SECTION I

INTRODUCTION

The design of a verifiably secure computer system must be expressed in a form that makes verification possible. The formal specification of a Multics security kernel presented in this volume and the next is sufficient to allow such a verification. This volume identifies and specifies the primary subsystem functions required to implement design choices described in the previous volume.

BACKGROUND

The specification presented in this volume and Volume III defines the top level input/output behavior of the Multics security kernel. The specification defines every available operation and every visible state of the hardware and software that will implement the kernel virtual machine. Since the specification is proven against the mathematical security model, it is known to define the function that must be fulfilled for a secure and correct implementation.

To unambiguously define kernel operation, a formal language has been adopted. The language can specify any computable process. By design, the language is non-procedural and hence is not meant to specify implementation requirements. Since several of the kernel principles depend on implementation properties (e.g., isolation), implementation conventions are required in addition to the formal language to completely define kernel operation.

The specification defines a kernel that is intended to be compatible with an existing system. Because of this constraint, a technique was required to allow the specification to define all the kernel functions and their correct operation in a manner consistent with the existing implementation. A method of interpreting the specifications for this purpose has been developed.

SYNOPSIS

In this volume, the storage system, process management, and external I/O facilities to be provided by the kernel are defined. In the next section, the specification technique, the language for specification and the conventions that govern the interpretation of the specifications, are described. The top-level kernel primary subsystem specifications and a detailed description of their meaning are

presented in the remaining sections. The secondary subsystems are presented in the third and final volume.

SECTION II

SPECIFICATION

The top level specification is a formal description of a finite state automaton intended to abstractly implement a reference monitor (as defined in [1], [2], and [3]) for the Multics system. The reference monitor requirements of verification imply a need for a rigorous and unambiguous specification and for identification of mediation and isolation mechanisms. The specification presented in this volume provides the necessary explicitness by adhering to a formal specification technique. The technique consists of a formal language and a set of conventions that restrict the interpretation and implementation of the specification.

Using the formal language, a kernel machine can be defined and proven to correctly implement the mathematical security axioms. Nevertheless, the abstract specification only defines a virtual machine. It obeys the isolation and mediation properties of a reference monitor by definition. It does not, however, define how these two properties are to be achieved in a physical implementation. They are instead specified by additional requirements.

FORMAL LANGUAGE

The form of the specification is derived from a technique developed by Parnas [4] and extended by Price [5]. A "Parnas specification" is a method of describing any computable process and thus any finite automaton. The state of the machine is embodied in a set of primitive "value" functions (V-functions), so called because when invoked they yield the last value assigned to them. The set of possible machine state transitions is defined by a set of "operate" functions (O-functions), so called because they define the operations that can be performed on V-functions.

The Parnas technique has three basic parts that allow a more concise specification. "Hidden" V-functions are introduced to provide a distinct separation of the information repositories from the information accessing functions. The Hidden V-functions are protected objects; therefore, they are defined to be inaccessible or "hidden". The primitive Hidden V-functions are the only information containers in the specification. They are the only functions that can be both observed and modified. To access them, the visible O- and V-functions must be used.

To "observe" the repositories from outside the kernel, the interface V-functions must be used. These interface functions may only be read. Their value is computed from the Hidden V-functions of the specification. For that reason, they are sometimes called "derived" or "mapping" V-functions as opposed to the "primitive" Hidden V-functions.

Similarly, only the interface O-functions may "modify" the kernel repositories. Their effect is stated as a set of assertions about the Hidden V-functions. Thus, the O- and V-functions make an initial partition of the allowed accesses to kernel information. In addition, through "exception" statements, these interface functions dictate the specific controls that will be enforced over each access to the repositories.

A special subset of O-functions, called OV-functions, provides for specification of indivisible operations. OV-functions return a value based on the results of the operation effected.

Three simple extensions that contribute to a more compact specification are strong typing and V- and O-function macros. The specification language is a strongly typed language, thus many of the argument validation checks are implicit.

V- and O-function macros support a macro mechanism that allows abbreviation in the specification. They are like visible functions but with no exceptions. A V-function macro may be used anywhere an expression might. Its value is the result of the derivation specified. An O-function macro may only be used in effects sections. Its effect is the result of the assertions specified.

SYNTAX AND SEMANTICS

The specifications are made explicit by expressing them in a well-defined language. The language is defined by a syntax (Appendix II) and a verbal semantics (below). The discussion of the language may be more meaningful if examined in parallel with the subsection entitled "THE MODULES" below.

A specification is defined to be a set of interdependent modules. The modules either correspond to the major subsystems of the kernel or are groups of functions separated for discussion purposes. Only the specification as a whole is complete and independent.

A module specification consists of nine parts: type, define, parameter, constant, Hidden V-function, V-function, O-function, OV-function, V-function macro, and O-function macro. The first four

parts are the basic definitions; they define the environment of a module. The last five parts are the functions; they define the meaning of a module.

Basic Definitions

The meaning of the basic module parts defined in the appendix is described below.

Types

The type part gives global type definitions that are used to declare the type of parameters and function values. Type definitions reduce the complexity of the module, without sacrificing generality, by providing strong typing. (In the implementation, type checking will be explicit.)

The syntax of the type definitions is adapted from the language PASCAL [6]. Although they can be explicitly defined, primitive types (Boolean, character, integer) are assumed to be well known.

Complex data types are built up from already defined types and type constructors. The "scalar" type constructor defines an ordered set by a list of constants. A subrange of a type is specified by the type definition followed by the first and last values of the range. (Subranges make sense only for partially ordered types). A "vector" is defined by the range of its selector (normally over the integers) followed by its component type. A "structure" type constructor is a list of field names and the types of the fields. The "set" type constructor constructs a set from a base type; the range of values defined is the power set of the base type.

Define

The define part is a list of module-global abbreviations whose meaning is the direct substitution of the definition text for the associated name, whenever it appears elsewhere in the module. The define abbreviations are simply a shorthand used to reduce the size of the specifications.

Parameter

The parameter part defines the formal parameters and quantified variables used in functions and defines their type by associating a type name or type definition with them. This section combines with the type section to provide strong typing. When functions are defined using these formal parameters, their argument types are implicitly defined.

Constant

The constant part lists invariant parameters of the specification whose actual value is not of interest. The constant names are used because they are more suggestive than the actual values. When a constant's actual value is not given, only its type and invariance are required for security. Constants will be constrained in the validation to have unique values.

The Functions

The body of each function describes what takes place when the function is invoked. Depending on the type of function, its body has up to four sections:

The "let" section is a list of function-local abbreviations. Their meaning is to simply substitute the definition text for the associated name, wherever it appears within the function.

The "exception" section contains boolean-valued expressions that have the value "true" when an error is detected. When true, exceptions are defined to abort the invocation of the function at once and return to the caller, indicating the reason for failure. These expressions are always evaluated in order. An "if" clause in an exception indicates that the statements of the "then" and "else" bodies are conditional exceptions. They are evaluated (as exceptions) only if the predicate is satisfied.

The "effect" section contains boolean assertions on V-functions that define transitions in the module state. The function is complete if and only if all the assertions are true. The meaning of the effect expressions is made unambiguous by using single quotes (') before names to represent the previous value associated with the name and unquoted names to refer to the associated value at the completion of the function. The ordering of these expressions is unimportant.

The "derivation" section is an expression that defines the value of a function. If the function is a structure-valued function, the derivation is a set of assertions defining each of the components of the structure.

All function definitions consist of a function name followed by a formal parameter list. The formal parameters are substituted for by the matching actual parameters in any particular invocation. All V-functions (including Hidden and OV-) have a result type, set off from the parameter list by a colon.

The five function parts of the module are described below.

Hidden V-function

The primitive Hidden V-functions are the module information repositories. They have no function body. They need no exceptions as they are not visible at the module interface. They may only be observed and modified by other module functions that are specified to do so correctly.

V-function

V-functions are the module interface "observe" functions. They do not have effects. In the specification all V-functions must be derived.

O-function

The O-functions are the module interface "modify" functions. They do not have a value. O-functions modify the Hidden V-functions of the specification to achieve their effect.

OV-function

OV-functions are equivalent to an O-function followed by a V-function without interruption. They have both effects and a value. The value is derived after the effects are completed.

V-function macro

V-function macros are simply value-computing macros with parameters. They consist of an optional let section and a derivation section. They are another shorthand that can be expanded mechanically in line.

O-function macro

O-function macros are also parameterized macros, but have an effect rather than a value. They consist of an optional let section and an effect section. They can only be invoked in effects sections of other O-functions, preceded by the keyword "effects_of". Like the other specification macros, they are a shorthand that can be mechanically expanded in line.

Semantic Conventions

The operators of the language have not been explicitly defined, because such a level of detail was not felt to be necessary. Only

well-known mathematical expressions are used and normal precedence holds, otherwise a comment is included.

The string constant "undefined" is special in meaning. When an entity is asserted to equal "undefined" it can be presumed to go to a state of inactivity such that, if and when reactivated, no previous history can be determined from the entity.

ISOLATION OF THE KERNEL

The implementation of the security kernel specification must be isolated if it is to meet the reference monitor requirements. Protection of the kernel from sabotage is a property that cannot be defined by Parnas specifications. Although a specification can prevent sabotage through the kernel interface, it cannot know what avenues will be opened by any particular implementation.

In the implementation of the Multics kernel a known and effective hardware protection mechanism is to be used to isolate the kernel. This mechanism will satisfy the model isolation requirement.

The mechanism to be used is the hardware ring mechanism of the Multics processor [7]. This mechanism and its use have been discussed in Volume I. Briefly, the kernel will be contained in the one or two innermost rings defined by the hardware and only a small set of predefined "gates" will allow controlled entry to this hardware domain. The kernel implementation will never allow any process to execute in the hardware kernel domain except when it has entered that domain through the controlled interface points. Similarly, the implementation will never allow processes to create segments in the hardware kernel domain except when legitimately in that domain by passing through a gate. Thus all kernel ring segments may only be created and accessed by a process when it has entered that ring through a gate and is executing verified code.

MEDIATION MECHANISMS

For the most part, the kernel specification defines specific interface functions for accessing objects. These functions ensure that access is correctly mediated because they only allow interpretive access to any object. The operations performed are carefully constrained.

There is a large class of functions, however, that must be implemented for compatibility, but whose operation degenerates to a simple process-local computation plus some combination of reads and writes on

behalf of the current process. Since the only security related operations of this class of functions are the accesses, it was felt that the kernel could be greatly simplified if only the restrictions on the accesses were specified. They would be in the form of generic read and write functions at the level of the process and would eliminate the need to specify the actual computation performed.

A verification issue is raised, however: the functions handled in this manner are required for compatibility and yet are now unspecified. This issue is resolved by a straightforward concept that uses a simple construction to satisfy both the security mapping and the compatibility requirements. This technique is described in detail in the next section entitled, "THE INTERPRETER CONCEPT".

THE MODULES

The top level specification consists of eight modules. Six describe the major subsystems of the kernel: Process Management, Storage System, External I/O, Reconfiguration, System Security Officer, and Initialization. (The last three are described in Volume III.) Each of these modules has a section devoted to its description. The seventh module, the Interpreter Support module, is described in the next section. The eighth module is the Common module. The Common module contains the parameters that are used to communicate between modules and the functions that are used by other modules without belonging to any one in particular. An explanation of the Common module provides a good example of the use of the specification language.

Common Basic Definitions

The Common type definitions are illustrated in Figure 1. Most of the types are self explanatory; a few require explanation or should be noted because of their impact on the other modules.

"access_level_type" is an embodiment of the access level defined in the mathematical model. It consists of two components: an integrity level, "il", and a security level, "sl". The integrity and security levels are of type "il_type" and "sl_type", both of which consist of a classification, "class", and a category, "cat".

"calendar_time_type" and "uid_type" are identical because the calendar time is used to generate unique identifiers (uid's).

"entry_type" is the name of a segment that uniquely identifies it within its parent directory. It is equivalent to the primary name of a segment in Multics.

```
  /*common type definitions*/

type
  class_type = scalar(unclassified, confidential, secret, etc.)

  compartment = scalar(nato, nuclear, china, etc.)

  category_type = set(compartment)

  il_type,
  sl_type = structure(class: class_type
                      cat: category_type)

  access_level_type = structure(il: il_type
                                sl: sl_type)

  uid_type,
  calendar_time_type = integer(0 to 2^time_length-1)

  char_string = vector(0 to string_size) of character

  entry_type = vector(1 to max_entry_size) of character

  ga_type = structure(seg: seg_type
                      offset: offset_type)

  machine_word_type = vector(0 to word_length-1) of bit

  offset_type = integer(0 to max_offset)

  principal_id_type = structure(user: char_string
                                project: char_string
                                tag: character)

  seg_type = integer(0 to max_seg_no)

  audit_log_type = structure(operation: char_string
                      op_access_level: access_level_type
                      user: principal_id_type
                      user_level: access_level_type)
```

Figure 1. Common Type Definitions

"ga_type" is a generalized address. For compatibility with Multics, it consists of a segment number, "seg", and an offset within the segment, "offset". Non-security related address information (e.g., effective ring number) is omitted.

"principal_id_type" is used to identify the owner of a process. It consists of the name of the owner, "user", his project name, "project", and an additional character to distinguish among his several processes, "tag".

"audit_log_type" is the definition of an audit log entry. It records the operation being audited, the level the operation occurred at, the user invoking the operation, and his access level.

Parameters

The Common parameters are illustrated in Figure 2. These formal parameters are used in functions in the common module and in other modules. The types of the parameters define the argument types of the functions they are used in. The common parameters are particularly important in that they establish a mechanism for arguments to be used consistently in all other modules.

Constants

The Common constants and definitions are illustrated in Figure 3. The constants are system hardware parameters or software imposed limitations that insure finiteness. The root of the directory hierarchy is also a constant since it must always be defined.

Definitions

The Common definitions (also Figure 3) represent widely used calculations that are given abbreviated names. In addition to reducing the size of the specification, the names are chosen to be mnemonic and thus improve the readability of the specification.

The abbreviation "Cur" permits convenient representation of all the attributes of the current process. "Cur.principal_id" expands to "Process(Cur_process).principal_id" which is the principal identifier of the current process.

The abbreviation "Branch" uses "Cur" to find the entry in the V-function Directory that contains the attributes of the segment "seg". "Dir_branch" is analogous to "Branch" except that it represents the entry containing the attributes of the directory "dir".

```
    /*  common parameters  */

parameter

  dir,
  seg: seg_type

  entry: entry_type

  dir_level,
  level,
  levela,
  levelb,
  seg_level,
  subject_level,
  object_level: access_level_type

  offset: offset_type

  principal_id: principal_id_type

  device_id,
  process_id,
  time: uid_type

  quota: integer

  drive_no: integer(0 to no_of_drives)
```

Figure 2. Common Parameters

```
    /*  common constants  */

constant

  max_entry_size: integer

  max_seg_no: integer

  string_size: integer

  time_length: integer = 54

  word_length: integer = 36

  max_offset: integer = 2^36

  root_uid: uid_type

  root_seg: seg_type

  no_of_drives: integer


    /*  common definitions  */

define

  Cur = Process(Cur_process);

  Branch = Directory(Cur.KST[Cur.KST[seg].dir].uid, Cur.KST[seg].entry);

  Dir_branch =
    Directory(Cur.KST[Cur.KST[dir].dir].uid, Cur.KST[dir].entry);
```

Figure 3. Common Constants and Definitions

These abbreviations will be more easily understood when they are used in functions.

Common Functions

The Common Hidden V-functions and V-function macros are illustrated in Figure 4.

The "Secure_" functions embody the access-level lattice relation checks required for the different possible access modes. The arguments to the functions are the access levels of a subject and object. The functions have the boolean value "true" if the access levels are in the correct relation, under the model axioms, to allow the access mode being checked for. Secure_read checks for observe access. Secure_alter checks for modify access. Secure_write checks for observe and modify access.

The Dominates function defines the access-level lattice relation according to the partial ordering of security and integrity levels. It is used to compute the "Secure_" functions and also to make direct lattice comparisons on levels.

The Unique_name function uses Current_calendar_time to define unique identifiers. This method is reasonable because the fine resolution of the clock (microseconds) and its long period (approximately 150 years) ensure the bit patterns it generates cannot be modulated and will be unique for the life of a system.

Audit_log is the primitive Hidden V-function that maintains the log entries. It is modified by an O-function macro (described below) each time an entry is made in the log. The audit entries may occur at any access level, but since Audit_log can only be read by a "system_high"[1] process, the *-property is not violated.

V-functions

Current_calendar_time (Figure 5) is a special function. It derives its value from the system hardware clock. It cannot be modified at the user interface but must be read by all, therefore the access level of Current_calendar_time is "system_low". Since V-functions are by definition read-only, no access check is required.

---

[1]"system_high" is the most priviledged access level in the MITRE model (the highest security level and lowest integrity level). It can observe, but never modify, any object. Its antithesis is "system_low". "system_high" and "system_low" are the join and the meet of the access lattice.

```
    /*  common Hidden V-functions  */


Hidden_V_function Audit_log(time): audit_log_type
    /*  audit log information  */


    /*  common V-function macros  */


V_function_macro Secure_read(subject_level, object_level): boolean

derivation
  Dominates(subject_level, object_level)


V_function_macro Secure_alter(subject_level, object_level): boolean

derivation
  Dominates(object_level, subject_level)


V_function_macro Secure_write(subject_level, object_level): boolean

derivation
  Secure_read(subject_level, object_level) &
  Secure_alter(subject_level, object_level)


V_function_macro Dominates(levela, levelb): boolean

derivation
  (levela.sl.class ≥ levelb.sl.class &
  levela.sl.cat ⊃ levelb.sl.cat) &
  (levela.il.class ≤ levelb.il.class &
  levela.il.cat ⊂ levelb.il.cat)


V_function_macro Unique_name: uid_type

derivation
  'Current_calendar_time;
```

Figure 4. Access Lattice Functions

```
    /*  common V_functions  */


V_function Current_calendar_time: calendar_time_type
  /*microseconds since 0000 GMT 1 January 1901*/

derivation
  "the current time"


  /*  common O-function macros  */


O_function_macro Audit(operation, access_level)

let
  time = Current_calender_time;

effect
  Audit_log(time).operation = operation;
  Audit_log(time).op_access_level = access_level;
  Audit_log(time).user = Cur.principal_id;
  Audit_log(time).user_level = Cur.access_level;


    /* common OV-functions */


OV_function Read_audit_log: audit_log_type

exception
  Cur.access_level ^= "system_high";
  Audit_log = "undefined";

effect
  Audit_log(min{(itime)(Audit_log(itime) ^= "undefined")}) = "undefined";

derivation
  'Audit_log(min{(itime)(Audit_log(itime) ^= "undefined")});
```

Figure 5. Time and Audit Functions

O-function Macros

The O-function macro Audit (also Figure 5) creates an entry in the audit log when it is invoked. It takes as arguments the operation and access level to be logged and enters them, along with the current user principal identifier and access level, indexed according to time. This O-function macro is invoked by Create_proc, Create_seg, Initiate, and Get_device to fulfill the military security policy requirements as discussed in Volume I and in [8]. Audit has no exceptions because it cannot be invoked at the kernel interface.

OV-functions

The OV-function Read_audit_log (also Figure 5) allows any "system_high" process to print the audit log created by Audit. Its value is the oldest entry in the log that has not yet been read.

Common Module -- Conclusion

The Common module has one O-function and only one special V-function. The only access check it makes is to ensure that the audit log is only read by a "system_high" process. Except for Audit_log, the Hidden V-functions are all derived and therefore have no access level associated with them. The access level of the information they reveal can be computed from the access levels of the information used in their derivations.

SPECIFICATION REVIEW

The specification language defines a framework for specifications and makes them more understandable through the uniformity it imposes. The common module sets the background from which the other specifications will work.

SECTION III

THE INTERPRETER CONCEPT

The interpreter concept is an abstraction technique used to reduce the amount of non-security information in the kernel specification. Since by definition the security model requires only the correctness of transfers of information between containers, the (sometimes complex) transformation of any single level data element is not a security issue. In particular, the single level nature of active subjects (processes) implies that process-local algorithmic manipulations of data need not be specified by the kernel. Only the transfers of data to and from each process need be specified.

The initial application of the interpreter concept was to avoid specifying as part of the kernel the myriad of details involved in the Multics hardware instructions. It was felt that all such functions could be broken into two disjoint parts: 1) any observations or modifications of data, and 2) all single level algorithmic manipulation of the observed data. Only the first part is security related, and thus, the specification would only include it. The correctness of the second part is not required for the security of the specification.[2]

TECHNIQUE

The interpreter concept is a means of visualizing the operation of a kernel machine that does not include extraneous functionality of single level computation. It allows a mapping between a minimal abstract machine and more complex concrete implementation.

The interpreter is defined to be an abstract program, outside the kernel, that runs on the kernel abstract machine. The interpreter conceptually provides an interface that defines a layer of the abstract machine outside the access control perimeter in a manner similar to the layers of abstract machines within the kernel.

---

[2]Note that although the security of the top level specification depends only on its obeying the properties of the mathematical security model, the implementation of the security kernel will depend on the correctness of any hardware instruction (algorithm) it uses. Nevertheless, the proof of correctness of an algorithm is a much simpler proof than the global behavior of a system. Therefore, in keeping with the principle of minimizing the kernel, it is not included in the specification.

To satisfy compatibility, the interpreter must present the interface normally available to programs. Because the interpreter is outside the specification kernel, however, it need not be specified for the verification of the specification as a correct interpretation of the access control model.

The kernel must provide at its access control perimeter all security-related functions required by interpreter operations. It must provide functions to create and access segments and state storage as required by the interpreter. The interpreter can then be envisioned as an abstract program that emulates all the non-security related functions required at the implementation interface.

APPLICATION OF THE INTERPRETER

The interpreter concept obviates the specification of three complex sets of functions. First, all the machine instructions can be embodied in a small number of generic functions. Second, the kernel functions that access only process-local information need not be specified. Third, the hardware ring mechanism can be ignored in the specification.

An indication was given above how the first benefit is achieved. The specification supports data observing and modifying functions and the interpreter defines the instruction-specific computation carried out on the observed data.

As an extension of this technique, the specification also supports "interpreter data", a separate process-local data base for each process, that is by definition, only accessible to that process. This data base can model such things as processor registers, virtual timers, faults, and signals; they only involve communication of the process with itself.

The final and possibly largest benefit is the elimination of the Multics ring mechanism from the specification. This application appears to be complicated by the use of rings to provide kernel isolation, but the isolation of the kernel is a separate issue from the security of the kernel addressed by the specification.[3] With regard to security, the ring mechanism available for general use is nothing more than process local communication using interpreter data.

---

[3]See the discussion in "ISOLATION OF THE KERNEL" in Section II and "VERIFICATION WITH THE INTERPRETER" below.

Within the constraints of the kernel, each process could set ring brackets on any segment it uses and interpret them in any manner without affecting security. To maintain compatibility in the specification, inter-process ring information can be thought of as being maintained in a data base outside the kernel that the interpreter accesses (subject to access control) on behalf of each process to emulate the appropriate ring checks.

INTERPRETER SUPPORT IN THE KERNEL

The interpreter module was not introduced as a kernel subsystem in Volume I because it is really a specification technique. The interpreter relies on other kernel subsystems for most of its support. It will add to functions in other modules any non-security operations required for compatibility (e.g., rings to Initiate in Storage Control). The interpreter module does have six interface functions of its own that will always have additional non-security effects. They do not change the state of the kernel but merely provide four generic functions for observing and modifying segment contents and two functions to access process-local interpreter data. These functions are the basic functions used by the interpreter to emulate the non-security hardware functions provided by the implementation (e.g., program execution, mathematical functions, ring checking).

"interpreter data" is a process-local state function. No operations on it are security related because it is isolated in a single process, but it must be a kernel object to ensure this isolation.

COMPATIBILITY WITH THE CURRENT MULTICS

The motivation for the interpreter is to establish the compatibility of certain kernel functions with current user functions. For the most part, the use of the interpreter functions is verbally defined but not specified. The narrative is intended only to indicate how the interpreter functions might be used to emulate a compatible interface. The correctness of the emulation is not a security issue: only the correctness of the kernel/interpreter interface need be shown. A proof of compatibility would require a specification of the emulation, but compatibility is only intended to be illustrated in this paper.

SPECIFICATION

Basic Definitions

The interpreter module is extremely simple. Most of its basic definitions come from other modules. In particular, since four of the interpreter functions are for accessing segments, the largest part of the interpreter basic definitions are in the storage control module. (A complete understanding of the interpreter functions depends on a familiarity with Section IV.) Figure 6 gives the specification of "interpreter_data_type". It is a vector of booleans whose length is implementation dependent.

The hidden V-function Interpreter_data (also Figure 6) is the repository for the interpreter data of each process. The access level of Interpreter_data is that of the process accessing it. Processes are only allowed to access their own interpreter data.

V-functions

The interpreter V-functions are given in Figure 7. The function Read_interperter_data provides observe access to the current process's interpreter data. It has no exceptions because it only allows access to process-local information.

Observing Segments

The functions Execute and Read provide the corresponding accesses to segments. Each function uses the storage control hidden V-function Inas to determine if the specified access is allowed to the segment and a check is made to prevent accessing beyond the end of a virtual segment.

Both Read and Execute observe segments. It is envisioned that the Execute function will be used by the interpreter to fetch instructions and that the Read function will be used to fetch operands. Two separate observe functions are required because discretionary access control differentiates between the two types of access. Since the interpreter can invoke these functions incorrectly, discretionary access can be subverted. This breach of discretionary controls is not an issue because discretionary access is known by its nature to be vulnerable to Trojan Horse attacks.

O-functions

The functions for modifying segments and interpreter data are given in Figure 8. The Write_interpreter_data function provides modify access to the process-local interpreter data. Like its

```
/* interpreter type definitions */

interpreter_data_type = vector(0 to *) of boolean

/* the size of the vector is implementation dependent */

/* interpreter parameters */


test: machine_word_type

interpreter_data: interpreter_data_type


/* interpreter Hidden V-function */


Hidden_V_function Interpreter_data(process_id): interpreter_data_type
  /* interpreter data for all processes */
```

Figure 6. Interpreter Module Basic Definitions

```
  /* interpreter V-functions */


V_function Read_interpreter_data: interpreter_data_type

derivation
  Interpreter_data(Cur_process);


V_function Execute(seg, offset): machine_word_type

let
  seg_uid = Cur.KST(seg).uid;

exception
  ^Inas(seg, "execute");
  offset > max_length;

derivation
  Data(seg_uid, offset);


V_function Read(seg, offset): machine_word_type

let
  seg_uid = Cur.KST(seg).uid;

exception
  ^Inas(seg, "read");
  offset > max_length;

derivation
  Data(seg_uid, offset);
```

Figure 7. Read and Execute Functions

```
  /* interpreter O-functions */

O_function Write_interpreter_data(interpreter_data)

effect
  Interpreter_data(Cur_process) = interpreter_data;



O_function Write(seg, offset, machine_word)

let
  seg_uid = Cur.KST(seg).uid;
  page = offset//page_size;
    /*  "//" is used to mean integer division  */

exception
  ^Inas(seg, "write");
  offset > max_length;
  (machine_word ≠ 0) & ^Page_allocated(seg_uid, page) &
     (Quota(seg).quota = Quota(seg).quota_used);

effect
  Data(seg_uid, offset) = machine_word;
  if ^'Page_allocated(seg_uid, page) & (machine_word ≠ 0)
     then effects_of Update_quota(seg, +1)
          Page_allocated(seg_uid, page) = "true";
  end;
```

Figure 8. Write and Test_and_set Functions

```
  /* interpreter OV-functions */


OV_function Test_and_set(seg, offset, test, machine_word): boolean

let
  seg_uid = Cur.KST(seg).uid;
  page = offset//page_size;

exception
  ^Inas(seg, "write");
  offset > max_length;
  (machine_word ≠ 0) & (^Page_allocated(seg_uid, offset)) &
    (Quota(seg).quota = Quota(seg).quota_used);

effect
  if 'Data(seg_uid, offset) = test
     then Data(seg_uid, offset) = machine_word;
  end;
  if ^'Page_allocated(seg_uid, page) & (machine_word ≠ 0)
     then effects_of Update_quota(seg, +1);
          Page_allocated(seg_uid, page) = "true";
  end;

derivation
  'Data(seg_uid, offset) = test
```

Figure 8. Write and Test_and_set Functions (concluded)

counterpart, Read_interpreter_data, it has no exceptions because it only allows access to process-local information.

Modifying Segments

The Write function provides "write" access to segments. "write" access is defined as both an observe and a modify access because information about the segment must be observed to successfully complete the modification. The exceptions check for the correct access using Inas. If a word is to be written to a previously unallocated page, a check must be made to ensure that there is sufficient quota to allocate that page.

The effects record the modification in the segment, and if the write was to a new page, the page is marked as allocated and the quota is adjusted. The write function thus automatically adds a page when required (when non-zero information is to be stored on it).[4]

The OV-function Test_and_set (also Figure 8) provides an uninterruptable observe and modify capability at the kernel interface. Its exceptions are the same as Write.

The effects of Test_and_set are conditional. If the present contents of the machine word addressed are equal to the specified test word, then the write is performed and the result is "true". Otherwise, no modification takes place and the result is "false". The derivation defines the result.

The Test_and_set function is required by the interpreter to implement locks and semaphores. Its design was chosen to be compatible with the corresponding hardware instruction on the current system.

VERIFICATION WITH THE INTERPRETER

Since the Multics kernel will be implemented on an existing hardware, the interpreter concept is complicated by a "correspondence" issue. In particular, the existing isolation mechanism that will be used does not provide sufficient granularity to separate kernel from non-kernel in the most desirable manner.

The Multics ring isolation mechanism does not define domains of hardware. Thus, in the implementation of the kernel, the hardware must be considered a single (kernel) domain, and initially, it would

---

[4]The deallocation of pages is discussed in the section on storage control (Section IV).

appear that the specification must include all hardware instructions. Otherwise, the implementation would include unspecified functions. Similarly, the software domains are, at a minimum, segment-sized. For efficiency, certain data bases required for compatible operation will have to be completely within one (kernel) domain. Again, it would appear that the specification must include all these data bases and operations on them.

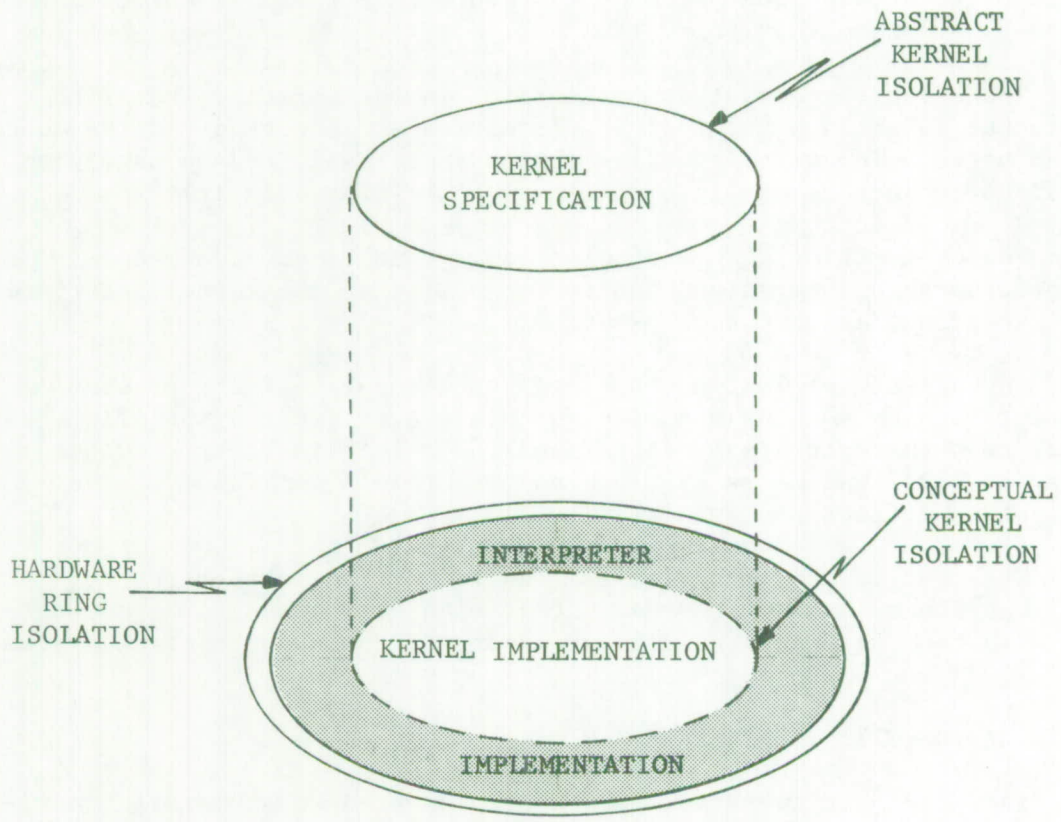

Figure 9. Implications of the Interpreter

Figure 9 depicts the implications of the interpreter concept. Although the interpreter concept makes it unnecessary to include any non-security functions in the specification the mismatch between the available implementation isolation mechanism and the desired "conceptual" kernel isolation implies that some non-security information will be in the kernel isolation domain in the implementation. In the figure, the broken circle represents the conceptual kernel isolation

boundary that corresponds to the specification interface. The double circle represents the physical kernel boundary that is enforced by the implementation domain mechanism (hardware rings).

It has been shown that the actual makeup of the interpreter is totally unrestricted with respect to verifying access control by the kernel specification. Nevertheless, the implementation is not unconstrained. Because we do not have a mechanism to enforce isolation of the interpreter implementation from the kernel implementation, some additional proof is required to demonstrate a correspondence.

The "brute force" proof would be to specify an interpreter and prove a correspondence in the same manner as the kernel. Note that only the existence of an interpreter specification need be shown, since it cannot affect the security of the kernel specification.

It is believed that a less difficult proof can be achieved for the interpreter implementation. It should be sufficient to demonstrate that the interpreter implementation only accesses data objects in a manner that corresponds to invoking a kernel function.

Because the specification of the interpreter cannot affect the proof of the security of the top level specification it is omitted. The choice of proof technique for the implementation of the interpreter will determine whether any specification is required. That choice is beyond the scope of this paper.

INTERPRETER REVIEW

The interpreter module describes the security-related segment and process-local data accessing functions. These functions do not change the kernel state; they only observe and modify information in the current access set. The interpreter itself is an unspecified set of abstract programs that use the basic kernel interface functions to demonstrate an interface compatible with the current system.

SECTION IV

STORAGE SYSTEM

The storage system kernel subsystem will implement the Multics virtual memory and control access to the information stored in it. Storage control functions provide controlled observation and modification of segment contents, creation and deletion of segments, and manipulation of segment attributes. In Volume I, an introduction was given to the current Multics storage system, the design decisions that defined the kernel storage system were reviewed, and the kernel storage system functions were introduced.

In this section, the details of the current Multics storage system (especially with regard to quota maintenance) will be given. With this background, the formal specification of the kernel storage system functions will be presented and the compatibility of the kernel with the existing system will be discussed.

CURRENT DESIGN -- DETAILS

Volume I discussed address space control, but omitted the details of the current quota mechanism because of their complexity. Quota controls must be discussed because they are required for both compatibility and security.[5]

In the previous Multics designs, even without the strict requirement of security, it was realized that usage of finite resources (storage in particular) must be deterministically regulated for correct operation. The current quota mechanism uses a construct called "quota cell" and the hierarchical segment structure to implement a straightforward control on the usage of secondary storage.

If the extension of demountable "volumes" of segments is ignored for the moment, the current system appears as in Figure 10: each directory segment has associated with it a quota cell consisting of a quota field and a quota-used field. The quota field tells how many records of secondary storage have been allocated to a segment (from the parent of the segment). The quota-used field tells how many records are presently in use by that segment and all its descendants. If the quota field of a quota cell is zero, it is called an "indirect"

---

[5]The detection of the finiteness of the physical system can be used as an information channel if not properly controlled.

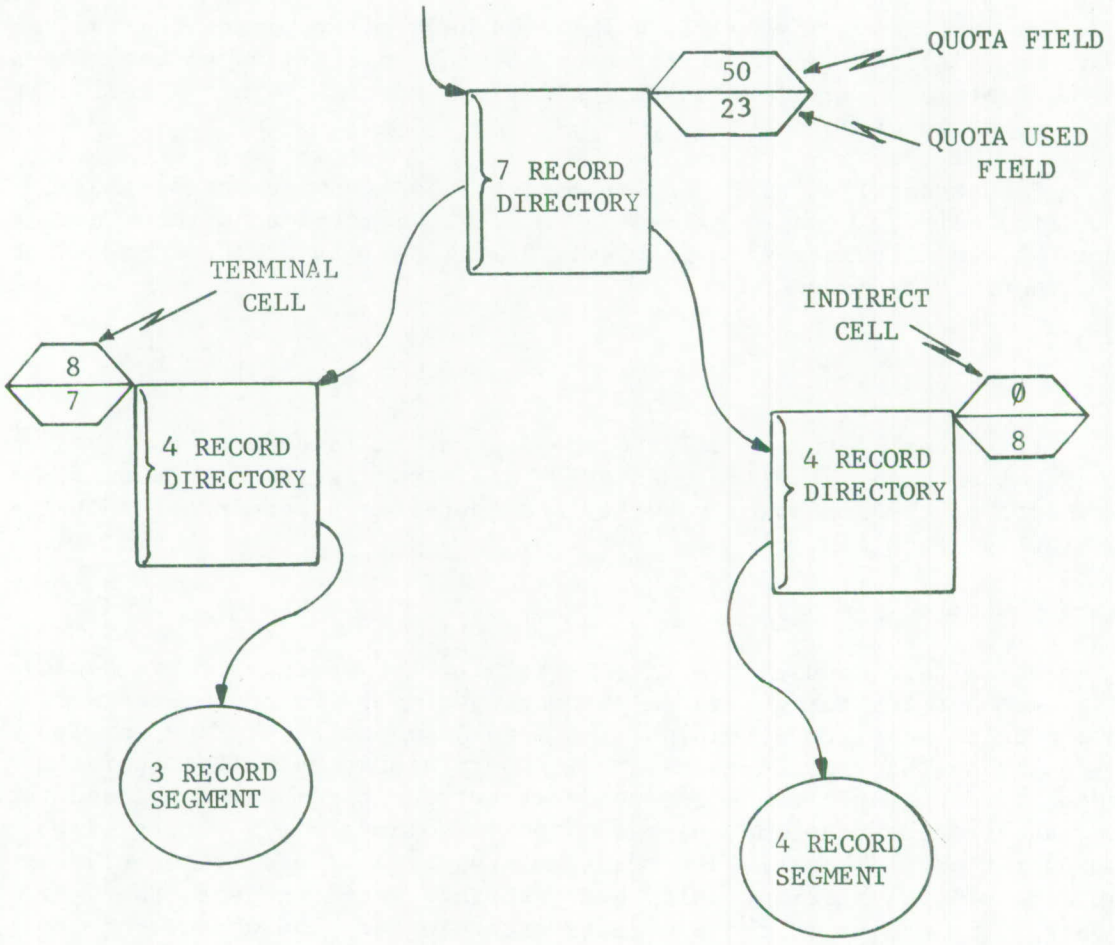

Figure 10. Quota Cell Mechanism

cell, and although it maintains an account of the quota used of the associated segment, the usage must actually be charged to the nearest ancestor with a non-zero quota (or "terminal" quota cell). Any intervening indirect cells will also keep account of the usage. No operation is ever allowed that would cause the quota-used of a terminal cell to exceed its quota.

Thus, if the quota cell at the root of the hierarchy is initialized to equal the storage quota of the system and quota is conserved when allocated to descendants (by charging to the quota used of the parent), usage can never exceed what is available. Note that a rather tricky situation occurs if a quota cell changes between terminal and indirect. The mechanism must ensure that the cell hierarchy is never changed in a way that makes it inconsistent with the above rules.

The extension of logical volumes is handled by having the equivalent of a "root" quota cell for each volume, initialized to the size of the volume. These cells are really just special terminal cells, in that they cannot be deleted.

Unfortunately, this quota mechanism, when constrained by the security policy, becomes extremely complex. Several adjustments are possible to alleviate this problem; their impact is discussed below as a compatibility issue.

THE KERNEL DESIGN

In Volume I, the kernel storage design was introduced and the constraints imposed by implementation decisions reviewed. In this subsection, the impact of security and correctness requirements on the design is discussed.

Access Control

Defining a secure interface is relatively straightforward because the elements of storage system correspond closely to the elements of the model, and by construction, the model's object hierarchy applies directly to the storage system's directory hierarchy. (This point is emphasized by the close correspondence between the model rules and the storage system functions, as was noted in Volume I.) Security (with respect to non-discretionary requirements) is achieved by associating an access level attribute with every storage system object, and controlling access to these objects accordingly. The objects of storage system are segments and segment attributes (including access level attributes). Since the security attributes of segment attributes will be the same as the security attributes of the segment itself or of the segment's parent, it is sufficient to associate an access level attribute with each segment.

The existing Multics access control list (ACL) mechanism implements need-to-know (discretionary) security.

Finiteness

When access control constraints are applied to the existing quota mechanism, it is revealed that the quota cells contain information at two access levels. To maintain access control, the quota field must be considered information at the level of the parent, and the quota-used field must be considered information at the level of the segment.

These constraints and the compatibility requirement of propagating the changes to a non-terminal quota cell through all intermediate indirect cells to the nearest terminal ancestor imply a very complex

specification. Taking direction from a Honeywell proposal, a change to the quota cell mechanism has been introduced that causes two incompatibilities, but greatly simplifies the specification of the quota mechanism and allows a flat-file system to be used as the underlying storage mechanism.

The change made is to eliminate indirect quota cells, thus eliminating the need for a complex updating system and limiting quota motion to the creation and deletion of quota cells. In addition, data segments are allowed to have their own quota cells, so that they can be upgraded to a level higher than that of their parent.

Correctness

To facilitate the proof of correctness of storage system, it is important to minimize the size and complexity of the storage system module. Two techniques have been employed to work towards this goal: 1) at the storage system interface segments will never be referred to by pathname; and 2) only those segments attributes that must be maintained by the kernel for security or compatibility are maintained.

The storage environment provided by the kernel has two aspects that we shall refer to as the system space and the local space. The system space is all of the on-line segments in the system, organized as a directory hierarchy. Any segment in the system space can be identified by its pathname -- a vector of length n where the first n-1 components identify the segment's parent directory, and the last component (the entry name, or entry for short) identifies the segment with respect to its parent. If the parent identification is implied by convention or supplied by some other means, then the entry name is sufficient to identify a segment in the system space.

The local space is a per-process address space. A process identifies segments in its address space by segment number (in the specification, "seg", or, if the segment must be a directory, "dir"). The segment number is a local name that is simply an integer index into a kernel-maintained, per-process data base. At the kernel interface, only two means of segment identification are provided: 1) by segment number (seg), if the segment is in the process's address space; and 2) by the parent directory's segment number and the segment's entry name (dir, entry), if the parent is in the process's address space.[6]

---

[6]This technique for identifying segments in a directory hierarchy was originally used in the design of a security kernel for the PDP-11/45 [9]. It has also been used in [10] as a technique for removing name space management from ring zero of the current Multics system, while leaving address space management in ring zero.

Table I

Kernel Segment Attributes

type (data or directory)
access level
access control list
volume identifier and sons-logical-volume-identifier
time-record product
quota given, quota, and quota used

Table I lists the segment attributes maintained by the kernel. Most of the attributes in Table I are taken from the set of entry attributes kept by the current Multics system [11]. All the attributes are essential for security except for the time-record product. It is felt that this accounting data must be kept by the kernel for efficiency.

To summarize briefly, the kernel design for storage system provides an environment similar to the current Multics virtual memory, but with a few basic differences. Security attributes are rigorously controlled and checked by the kernel. At the storage system interface, segments in the hierarchy cannot be identified by pathname, and only a subset of the current segment attributes is maintained by the kernel.

COMPATIBILITY WITH THE CURRENT MULTICS

One of the requirements of this kernel design is to maintain a high degree of compatibility with the current Multics user interface. In this subsection we consider the role of the non-kernel supervisor with respect to the storage system. Since Bratt has demonstrated that pathnames can be removed from the current Multics ring 0 without any loss of compatibility [10], it remains for us to consider the support of segment attributes removed from the kernel and the impact of security on the quota mechanism.

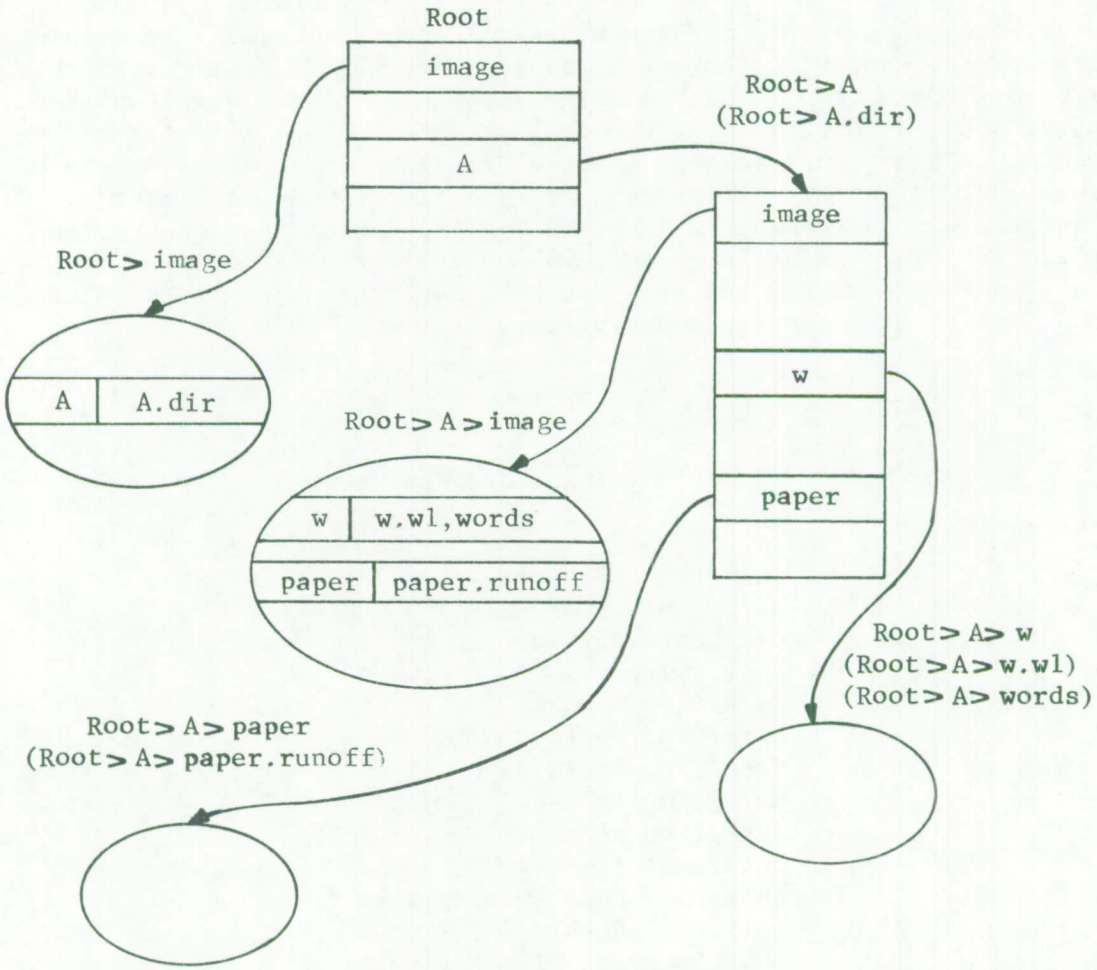

Figure 11. Hierarchy with Directory Images

Directory Images

The interpreter concept defines a technique that can also be used by the supervisor, outside the kernel ring, to support the removed segment attributes. A special data segment called the directory image can be associated with each directory to contain the non-kernel attributes. Since the association of directory images with directories is made outside the kernel, to the kernel, directory images are indistinguishable from other data segments. The supervisor can make the separation of attributes in the directories and directory images invisible to higher levels of software. In particular, the supervisor can maintain the current Multics user interfaces with the file system.

Since the Root's directory image (like all other segments) must be inferior to the Root, it is reasonable to have each directory image

immediately inferior to its associated directory. It will be necessary for the supervisor to establish a convention for getting to the directory image from the directory. Figure 11 shows a simple hierarchy with two directories and two images. At the user interface each segment in the figure (except for the directory images) has multiple names, but at the kernel interface only a single name is allowed. The alternate names are kept in the images. Attributes that can be kept in the image and maintained by the supervisor are listed in Table II. (Note that the image attributes are unknown to the kernel; they are shown only to illustrate compatibility.)

Table II

Directory Image Attributes

author
bit count
bit count author
copy switch
date/time dumped
date/time entry modified
date/time modified
date/time salvaged
date/time used
gate/call limiter
initial access control list
maximum length
multi-segment file indicator
multiple names
ring brackets
safety switch
unique identifier

The use of the directory image will impact the structure of the hierarchy. The compatible-hierarchy constraint[7] allows segments inferior to a given directory to be at any access level greater than or equal to the level of the directory. Since attributes at the level of the directory and attributes at the level of each segment immediately inferior to the directory will be kept in the directory image, the supervisor should impose the additional constraint that all data

[7]The compatible-hierarchy constraint (termed compatibility constraint in [12]) requires the access level of segments to be non-decreasing as one moves away from the Root.

segments be at the same access level as their parent if the image is to be a single segment. Directories have their own image and thus need not be at the level of the parent. The special case of a directory at a level greater than its parent will be referred to as an "upgraded directory" [13].

Whether the image attributes are actually maintained in separate segments is an implementation decision. The interpreter concept allows either approach to be taken. The decision must be based on a tradeoff between performance and ease of proof.

The Quota Mechanism

The incompatibilities introduced by the change to the quota mechanism are minimal. First, a segment, once created, cannot change its terminal or non-terminal status. (This option is seldom exercised in the present system.) Second, the quota used by any subtree of the hierarchy that is headed by a non-terminal node is no longer maintained. It can be estimated if necessary by scanning the subtree. If the quota of a subtree must be known accurately, it can be headed by a terminal node.

By allowing a quota cell to be associated with a single data segment, that data segment can be upgraded to an access level higher than its parent. Since the quota cell is associated with only a single data segment (as opposed to the case of a directory), when the segment is deleted the quota can be returned to the parent without any security complications.

The static nature of quota cells implies segments cannot be dynamically upgraded. The solution to the problem is to simulate the upgrade by copying the segment to the higher level and deleting the old copy.[8]

---

[8]For efficiency reasons, the new storage system requires all directories to be on the (always mounted) root volume. This segregation of directories and segments introduces additional complication in implementation but does not impact the specification. This complication can be handled by envisioning two parallel quota trees, one for directories and one for data segments. The accessing functions for each tree would be as specified; however, the trees would be initialized differently according to the separate volume organizations they must control.

Message Segments

Since message segments must be controlled by the kernel in the same manner as all other data objects, it was decided they should be implemented outside the kernel using data segments. Multi-level message segments can be simulated by the message system with a directory of single level data segments, one for each level of messages that can occur.

Backup and Retrieval

Backup and retrieval will be greatly simplified by following the proposal for the new storage system. The backup daemon will be driven by a list of segments that have been modified, rather than scanning the hierarchy. This daemon will thus operate in a manner similar to the other daemons, receiving its requests via the message system. A single daemon running at system high is sufficient. Backed up segments will be at system high so they will have to be donwgraded to be restored, but since human intervention is required for this operation anyway, downgrading should not be much of an additional burden.

SPECIFICATION

This subsection gives a top-level specification of the kernel's storage system subsystem. Before proceeding, we will discuss the role of unique identifiers (uid's) in this specification.

Unique Identifiers

The functions available at the storage system interface support only one means of segment identification -- by segment number. A segment number is a process-local name for a segment in a process's address space. There are actually three cases for segments observed or modified by storage system functions: 1) a segment identified by segment number; 2) a segment immediately inferior to a segment identified by segment number (in this case the interface function includes a parameter that uniquely identifies the inferior segment with respect to its parent directory); and 3) a segment superior to a segment identified by segment number.

Since two different processes can have different segment numbers for the same segment, and the contents of segments must be more permanent than temporary segment numbers, the primitive V-functions that represent the contents of segments cannot be parameterized by segment number. Instead, permanent, system-global unique identifiers (uid's) are used. Thus, in the storage system specification, each segment is identified by a uid.

It is important to remember that the specification uids are a mechanism for unambiguously referencing segment contents. As such, they roughly correspond to physical addresses (which are unique, but reused), not to the uid segment attribute maintained by the current Multics.

Basic Definitions

Type definitions used in storage system are listed in Figure 12. There are two different types of segments, data segments and directory segments. (Message segments are implemented using data segments.) Access modes are the different ways in which segments can be accessed. The access modes used in storage system, the segment types to which they apply, and the correspondence of access modes with the primitive access modes observe and modify is shown in Table III.

Table III

Storage Control Access Modes

data	read	observe
	execute	
	write	observe and modify
	enter	modify
directory	status	observe
	modify	observe and modify
	append	

Branches

A branch is the element of a directory that contains the attributes of a segment; "branch_type" defines the components of a branch, most of which should be self-explanatory. Each element of an ACL has two primary components, a principal identifier and a (possibly null) set of access modes. A description of the current Multics ACL mechanism is given in [14]. In general, the kernel mechanism directly supports the current features; exceptions will be explicitly identified.

```
    /* storage control type definitions */

type

  seg_type_type = scalar ("data", "directory")

  access_mode_type = scalar ("read", "write", "execute",
     "enter", "status", "modify", "append")

  branch_type = structure
     (uid: uid_type
     type: seg_type_type
     access_level: access_level_type
     ACL: ACL_type
     last_acli: integer
     vol_id, sons_vol_id: uid_type
     quota, quota_given, quota_used, TRP: integer
     LT: calender_time_type)


  ACL_type = vector (1 to max_acli) of structure
     (id: principal_id_type
     mode: set (access_mode_type))


  acli_type = scalar (1 to max_acli)


  message_type = vector (0 to max_message_size) of machine_word_type
```

Figure 12. Storage Control Type Definitions

"last_acli" is the current number of elements in a segment's ACL. "vol_id" is the logical volume on which the segment resides; "sons_vol_id", defined only for directories, is the logical volume on which all non-directory sons of the directory are stored. "LT" is the last time that the segments "TRP", time-record product, has been updated.

Quota

The quota entries deserve special mention since they implement quota cells. "quota_given" is the quota that was given to the segment at creation. It thus records the total quota allocated to the subtree headed by the segment. ("quota_given" was introduced as an aid to the verification. It could be eliminated from the implementation with an appropriate equivalence proof.)

"quota" is initially the same as "quota_given", but it will be deducted from if the segment is a directory and allocates some of its quota to a child.

"quota_used" is the total quota being used by all the descendents of the segment who charge to the quota cell of the segment (do not have their own quota). Note that if the segment does not have a quota cell, all the quota fields are undefined.

Parameters and Constants

The definitions of parameters (including quantified variables) and constants are shown in Figure 13. The meaning of the parameters will be obvious when they are used in functions. "max_length" is the maximum size in words of a directory or segment. "max_acli" is the maximum number of elements in a single ACL. "page_size" is the size of a record. "max_message_size" is the maximum length of a message in words.

Hidden V-functions

Storage control contains five primitive V-functions that record the state of the storage system -- two for segment contents, two for demountable volumes, and one for page control (see Figure 14). Data segments contain machine words; each word is identified by an offset. Directory segments contain branches; each branch is identified by an entry (name).

The function LVRF (logical volume registration file, a term taken from the current Multics implementation) contains a structure for each demountable volume known to the system. Logical volumes are identified by "vol_id", a unique identifier. The information that the

```
    /* storage control parameters */

parameter

  term_seg,
  base_seg: seg_type
  seg_uid,
  dir_uid,
  vol_id,
  sons_vol_id: uid_type
  new_entry: entry_type
  access_mode: access_mode_type
  access_modes: set (access_mode_type)
  machine_word: machine_word_type
  acli: acli_type
  quota_change,
  ru_change: integer
  type: seg_type_type
  msg_value: message_type
  new_quota: non-negative integer

    /* and quantified variables */

  iseg: seg_type
  ioffset: offset_type
  ientry: entry_type
  iacli: acli_type
  iuid,
  iprocess_id: uid_type
  idrive_no: integer (0 to no_of_drives)
  itime: calender_time_type



    /* storage control constants */

constant

  max_length: offset_type
  max_acli: integer
  page_size: integer
  max_message_size: integer
```

Figure 13. Storage Control Parameters and Constants

```
    /* storage control hidden V-functions */


Hidden_V_function Data(seg_uid, offset): machine_word_type
    /* contents of data segments */


Hidden_V_function Directory(dir_uid, entry): branch_type
    /* contents of directory segments */


Hidden_V_function LVRF(vol_id): structure
     (mounted: boolean
     access_level: access_level_type
     quota: integer)


Hidden_V_function Drive(drive_no): uid_type;
    /* status of disk drives */


Hidden_V_function Page_allocated(seg_uid, page): boolean
    /* status of segment pages */
```

Figure 14. Storage Control Primitive V-functions

kernel must maintain about each volume that exists is: its status (mounted or demounted), its access level, and the amount of available storage on it (quota).

The other function supporting demountable logical volumes is Drive_status; it identifies the logical volume mounted on each disk drive used for mounting demountable volumes.

The function Page_allocated keeps track of whether or not a page of a segment has been allocated. (A page is allocated when it has not been allocated and some non-zero information will be written in it. It is assumed that the implementation will not allocate a page that is all zero.) While the top level is not concerned with the details of page control, it must know which pages have been allocated and are thus charged to a quota cell. For determinism, pages must be deallocated by a kernel function, rather than automatically.

V-function Macros

Segment Access Checking

Most of the functions at the storage system interface involve segment accessing. All of these functions invoke the Inas (in address space) function (Figure 15) in their exception section to ensure that the user has the right, with respect to all security requirements, to access the segment in the desired mode. The parameters of Inas identify a segment and an access mode.

To access a segment the user must have the segment in his address space (CUR.KST(seg) ≠ "undefined") and he must have discretionary access rights in the appropriate mode. Also, as a function of the access mode, he must have an appropriate access level. Finally, the segment's type must be consistent with the access mode.

Although any access of a segment inferior to the Root implies an indirect observation of the Root, Inas does not allow direct access to the Root. Access to the Root can be handled in at least three different ways: 1) the Root can be fixed at system initialization; 2) special case checks for Root accesses can be added to Inas; or 3) distinct functions for accessing the Root can be specified. The current kernel design uses the first.

Specifications for the Access_permission function and a supporting function, Acle_apply, are shown in Figure 16. Access_permission checks a segment's ACL and indicates whether or not a user has a particular access right to a segment. Since more than one element in an ACL can apply to a user, Access_permission always uses the first one that does apply. Acle_apply determines if a particular ACL

```
    /* storage control V-function macros */


    /* V-function macros used primarily in exceptions */


V_function_macro Inas(seg, access_mode): boolean

let
  seg_type = Branch.type;
  seg_access_level = Branch.access_level;
  may_read = Secure_read(Cur.access_level, seg_access_level);
  may_write = Secure_write(Cur.access_level, seg_access_level);
  may_alter = Secure_alter(Cur.access_level, seg_access_level);

derivation
  if(Cur.KST(seg) ≠ "undefined")
     then Access_permission(seg, access_mode) &
            [caseof access_mode
               ("read" | "execute")! may_read & (seg_type = "data");
               "write"! may_write & (seg_type = "data");
               "enter"! may_alter & (seg_type = "data");
               "status"! may_read & (seg_type = "directory");
               ("modify" | "append")! may_write & (seg_type = "directory");
            end] &
            Branch.uid ≠ root_uid
  end;
```

Figure 15. Inas Function

```
V_function_macro Access_permission(seg, access_mode): boolean

let
  acli = min{iacli|Acle_apply(seg, iacli)};

derivation
  (∔iacli)(Acle_apply(seg, iacli)) &
     (access_mode ϵ Branch.ACL(acli).mode);



V_function_macro Acle_apply(seg, acli): boolean

derivation
  ((Branch.ACL(acli).id.user = Cur.principal_id.user) |
     (Branch.ACL(acli).id.user = "*")) &
  ((Branch.ACL(acli).id.project = Cur.principal_id.project) |
     (Branch.ACL(acli)~.id.project = "*")) &
  ((Branch.ACL(acli).id.tag = Cur.principal_id.tag) |
     (Branch.ACL(acli).id.tag = "*"));
```

Figure 16. Access_permission and Acle_apply Functions

element applies to a user. An element applies if each of the three fields in it's principal identifier component match the user's principal identifier or indicate "don't care" (a "*").

Quota Functions

Four functions associated with the quota mechanism are shown in Figure 17. NPFA and NPFB tell whether a new page would have to be allocated if an ACL element or branch were added to a directory. Their value is determined by checking if the offset the new branch or element would be stored at is in a page that has not been allocated.

"QC_ptr" identifies the segment whose quota cell is to be charged to by the input segment. The quota cell identified is that of the nearest (hierarchical) ancestor with a quota cell. The "Quota" function returns the quota and quota used attributes of the quota cell indicated by "QC_ptr".

Implementation Functions

The specification of functions that modify directory segments includes exception clauses that check for quota and segment overflow, and effect statements that conditionally update the directory segment's quota attributes. These exception and effect statements are included because the modification can cause the directory segment to change size, and this change can be observed at the user interface. The ability to observe this change is a security consideration.

Whether or not a directory segment actually grows by a page as the result of a particular function call depends to a large extent on low level implementation details of encoding and storage allocation within the segment. To avoid specifying implementation details, we have defined a set of functions that indicate where in a directory segment a new branch or ACL element is stored, without identifying how this value is actually determined. These functions are listed in Figure 18. Note that they specify precisely what is observed in determining their value.

The function Message_offsets is specified to support the O-function Enter_msg. It will be explained in the discussion of Enter_msg.

Verification Functions

The last two storage V-function macros (Figure 19) are included as an aid to the verification. Because directories and the logical volume registration file (LVRF) are multi-level information structures, information at several levels can be observed when checking for the existence of an entry. A special proof must be made to show that

```
    /* other V-function macros */


V_function_macro NPFA(dir_uid, entry, principal_id): boolean
    /* Need Page For (new) ACL element */

derivation
  ^Page_allocated(dir_uid,
    Acle_offset(dir_uid, entry, principal_id)//page_size);



V_function_macro NPFB(dir_uid, entry, type): boolean
    /* Need Page For (new) branch */

derivation
  ^Page_allocated(dir_uid,
    Branch_offset(dir_uid, entry, type)//page_size);



V_function_macro QC_ptr(seg): structure(qc_dir_uid: uid_type
                                        qc_entry: entry_type)

derivation
  if Branch.quota_given ≠ 0
     then qc_dir_uid = Cur.KST(Cur.KST(seg).dir).uid;
        qc_entry = Cur.KST(seg).entry;
     else QC_ptr(Cur.KST(seg).dir);
  end;



V_function_macro Quota(seg): structure(quota, quota_used: integer)

let
  qc_dir_uid = QC_ptr(seg).qc_dir_uid;
  qc_entry = QC_ptr(seg).qc_entry;

derivation
  quota = Directory(qc_dir_uid, qc_entry).quota;
  quota_used = Directory(qc_dir_uid, qc_entry).quota_used;
```

Figure 17. Quota Functions

```
    /* Implementation V-function Macros */

V_function_macro Acle_offset(dir_uid, entry, principal_id): offset_type
    /* offset in segment for new ACL element */
  observes
  (¥ientry)(Entry_defined(dir_uid,ientry) &
    Directory(dir_uid,ientry).ACL)


V_function_macro Branch_offset(dir_uid, entry, type): offset_type
    /* offset in segment for new branch */
  observes
  (¥ientry)(Entry_defined(dir_uid,ientry) &
    Directory(dir_uid,ientry).ACL)


V_function_macro Dir_page_exists(dir_uid, page): boolean
    /*  is directory page all zero?  */
  observes
  (¥ientry)(Entry_defined(dir_uid,ientry) &
    Directory(dir_uid,ientry).ACL)


V_function_macro Message_offsets(seg_uid, msg_value): set(offset_type)
   /* offsets in segment for new message */
  observes
  (¥ioffset)(Data(seg_uid, ioffset))
```

Figure 18. Implementation Functions

```
V_function_macro Entry_defined(dir_uid, entry): boolean

derivation
  Directory(dir_uid, entry).uid ≠ "undefined";


V_function_macro LV_defined(vol_id): boolean

derivation
  LVRF(vol_id).access_level ≠ "undefined";
```

Figure 19. Storage Verification Functions

no information is incorrectly revealed. By defining the two mapping V-functions "Entry_defined" and "LV_defined", these proofs need only be made once (see [15] for details).

V-functions

Two interface V-functions are provided for observing contents of directory segments. The accessing of data segments is provided for by the interpreter module.

The Seg_attributes V-function (Figure 20) returns all segment attributes that are at the level of the segment's parent directory. Thus, the security requirement is that the user must have "status" (i.e., read) access to the directory -- his access level must dominate the directory's.

The Seg_side_effect_attributes V-function returns the segment attributes that are set as a side effect of modifying the segment. These attributes are "TRP" (time-record product), quota, and "quota_used". Since the access level of a segment may dominate its parent's, checking "status" access for the parent is not sufficient. Any user at the level of a segment may modify it (subject to discretionary access controls), and therefore, may modify these "side effect" attributes. Thus, this function requires that the user have "Secure_read" access to the segment -- his access_level must dominate the segment's. The kernel does not maintain any attributes that are set as a side effect of observing the segment. If it did, the access level of these attributes would have to be at "system_high".

O-functions

All storage system functions that correspond to model transition functions are O-functions. In addition, there are two O-functions for mounting and demounting logical volumes, a message function, and two quota functions that do not cause security transitions. Before describing these O-functions, the single O-function macro used in storage system will be discussed.

The O-function macro Update_quota is used to specify the Multics quota mechanism (Figure 21). This function has two arguments, the segment number of a segment whose size is changing by at least one record and the change in the quota. The quota change can be the result of two actions. It changes when the segment itself changes size. It also changes when quota is transferred between segments. For example, in the Create_segment function the quota_change is equal to the change in size of the parent directory plus the quota given to the newly created segment.

```
    /* storage control interface V-functions */


V_function Seg_attributes(dir, entry): structure
                (type: seg_type_type
                access_level: access_level_type
                ACL: ACL_type
                last_acli, quota_given: integer)

let
  dir_uid = Cur.KST(dir).uid;

exception
  ^Inas(dir, "status");
  ^Entry_defined(dir_uid, entry);

derivation
  type = Directory(dir_uid, entry).type;
  access_level = Directory(dir_uid, entry).access_level;
  ACL = Directory(dir_uid, entry).ACL;
  last_acli = Directory(dir_uid, entry).last_acli;
  quota_given = Directory(dir_uid, entry).quota_given;;


V_function Seg_side_effect_attributes(dir, entry): structure
                  (TRP, quota, quota_used: integer)

let
  dir_uid = Cur.KST(dir).uid;

exception
  ^Inas(dir, "status");
  ^Entry_defined(dir_uid, entry);
  ^Secure_read(Cur.access_level, Directory(dir_uid, entry).access_level);

derivation
  TRP = Directory(dir_uid, entry).TRP;
  quota = Directory(dir_uid, entry).quota;
  quota_used = Directory(dir_uid, entry).quota_used;
```

Figure 20. Seg_attributes and Seg_side_effect_attributes Functions

```
  /*  storage control O-function macros  */


O_function_macro Update_quota(seg, quota_change)

let
  qc_dir_uid = QC_ptr(seg).qc_dir_uid;
  qc_entry = QC_ptr(seg).qc_entry;
  delta_time =
    'Current_calendar_time - 'Directory(qc_dir_uid, qc_entry).LT;
  delta_TRP = delta_time*'Directory(qc_dir_uid, qc_entry).quota_used;

effect
  Directory(qc_dir_uid, qc_entry).quota_used = *+quota_change;
  Directory(qc_dir_uid, qc_entry).TRP = *+delta_TRP;
  Directory(qc_dir_uid, qc_entry).LT = 'Current_calendar_time;
```

Figure 21. Update_quota Function

Update_quota is employed in every O-function that modifies the contents of segment, because whenever a segment is modified, its storage requirements can change. The effects of Update_quota are straightforward: the "quota_used", "TRP", and "LT" attributes of the quota cell to which the segment is charging are updated.

Demountable Volumes

Before a process can access a segment stored on a demountable volume, the process must use the Mount O-function (Figure 22) to request that the volume be mounted, even if some other process has already mounted the volume. (This requirement comes from the current Multics design for demountable volumes.)

The exceptions for the Mount O-function prevent a process from mounting a volume if that process already has the volume mounted or the volume does not exist. If the volume is already mounted, then the process state information is updated. Otherwise, a request is entered to have the volume mounted. (The process state information will be updated by the mounter when the mount is complete. The mount requests will be handled by the system security officer interface, discussed in Volume III.)

A process can demount a volume if it has the volume mounted and it does not have any segments on the volume initiated. The effects of Demount are to remove the volume from the mount list of the process and to cancel a pending mount request by that process for the volume. (The volume will only be physically demounted when no one has the volume logically mounted and the mounter needs the drives it is assigned to.)

Initiation, Termination, and Revocation

A process can only access a segment in its own address space. The Initiate O-function moves a segment into a process's address space (as described by the known segment table -- see Section V) and associates a segment number with it. Terminate performs the inverse operation (see Figure 23). These functions roughly correspond to the model transition functions Get access and Release access. They correspond roughly, because Initiate does not check discretionary access, and it does not determine the specific access modes that the process can use with respect to non-discretionary security requirements. We specify the enforcement of the specific discretionary and non-discretionary requirements for each access attribute in the functions that perform the accesses.

A segment is initiated in terms of its parent directory. A segment can be initiated only if the segment number is not in use,

```
    /* storage control O-functions */


O_function Mount(vol_id)

exception
  Cur.mount_list(vol_id);
  ^LV_defined(vol_id);

effect
  if LVRF(vol_id).mounted
     then Cur.mount_list(vol_id) = "true";
     else if ^(∔ itime)(Mount_request(itime).process = Cur_process &
                        Mount_request(itime).vol_id = vol_id);
             then Mount_request('Current_calendar_time).process =
                                                          Cur_process;
                  Mount_request('Current_calender_time).vol_id = vol_id;
        end;
  end;



O_function Demount(vol_id)

exception
  (∔ iseg)(Cur.KST(iseg).vol_id = vol_id);

effect
  if 'Cur.mount_list(vol_id)
     then Cur.mount_list(vol_id) = "false";
  end;
  if (∔ itime)(Mount_request(itime).process = Cur_process &
               Mount_request(itime).vol_id = vol_id)
     then Mount_request(itime) = "undefined";
  end;
```

Note: these functions are completed by the trusted functions described in Volume III.

Figure 22. Mount and Demount Functions

```
O_function Initiate(dir, entry, seg)

let
  dir_uid = Cur.KST(dir).uid;
  vol_id = Directory(dir_uid, entry).vol_id;

exception
  Cur.KST(seg) ≠ "undefined";
  Cur.KST(dir) = "undefined";
  Dir_branch.type ≠ "directory";
  ^Entry_defined(dir_uid, entry);
  (^Secure_read(Cur.access_level,
    Directory(dir_uid, entry).access_level) &
    (Directory(dir_uid, entry).type = "directory"));
  ^Cur.mount_list(vol_id);

effect
  Cur.KST(seg).dir = dir;
  Cur.KST(seg).entry = entry;
  Cur.KST(seg).uid = ´Directory(dir_uid, entry).uid;
  Cur.KST(seg).vol_id = vol_id;
  effects_of Audit("Initiate", Directory(dir_uid, entry).access_level);
```

Figure 23. Initiate, Terminate, and Revoke_access Functions

```
O_function Terminate(seg)

exception
  Cur.KST(seg) = "undefined";
  (∔iseg)(Cur.KST(iseg).dir = seg);

effect
  Cur.KST(seg) = "undefined";



O_function Revoke_access(dir, entry)

let
  dir_uid = Cur.KST(dir).uid;
  seg_uid = Directory(dir_uid, entry).uid;

exception
  ^Inas(dir, "modify");
  Directory(dir_uid, entry).type ≠ "data";

effect
  (¥iprocess_id, iseg)
    if ´Process(iprocess_id).KST(iseg).uid = seg_uid
       then Process(iprocess_id).KST(iseg) = "undefined";
    end;
```

Figure 23. Initiate, Terminate, and Revoke_access Functions (concluded)

its parent directory has been initiated, the segment exists, and the volume on which the segment is stored is mounted.

For each initiated segment the kernel keeps in the known segment table (KST) the segment number of its parent directory (dir) and the name of the segment with respect to its parent (entry), so that given a segment number, it can easily access that segment's branch. The kernel also keeps the segment's uid and volume id attributes in the KST. Although this information can be found in the branch, it simplifies other kernel functions to copy it into the KST at initiate time. Initiate also adds an entry to the audit log.

There are no security controls on the Terminate function, the kernel function that corresponds to the model's Release access transition function. The kernel does, however, prevent a directory from being terminated if inferior segments are still initiated, so that it is always possible to access an initiated segment's branch, and to find the quota cells in the hierarchy.

The Revoke_access function allows one process to terminate another process's access to a segment, presumably because the first process has modified some attribute of the segment that it wishes to propagate by forcing reinitiation of the segment. (e.g., For compatibility, a change to the ring brackets of a segment must be propagated immediately before any accesses are allowed to the segment.)

This function is included in the kernel specification because the access revocation is a form of communication from the first process to the second. The security policy requires that the level of the revoking process be less than or equal to any "revokee" processes. This requirement is satisfied by checking for modify access to the parent. It is sufficient to allow this function to be used for data segments, since directory segments are only interpretively accessed by non-kernel software. (The correctness of interpretive accesses is checked at each access.)

Creating and Deleting Segments

The kernel functions Create_segment and Delete_segment correspond directly to the model functions create object and delete object. The arguments to the Create_segment function identify a directory segment that will be the new segment's parent, a name for the new segment, and its type, access level, quota, and sons-volume-id-attributes (see Figure 24). The sons-volume-id-attribute is only meaningful if the new segment is a directory; if it is, and the sons-volume-id is different than the parent's sons-volume-id, then the new segment is a "master directory".

```
O_function Create_segment(dir, entry, type, access_level, new_quota,
                                                    sons_vol_id)

let
  dir_uid = Cur.KST(dir).uid;
  seg_uid = Directory(dir_uid, entry).uid;
  branch_page = Branch_offset(dir_uid, entry, type)//page_size;
  dir_access_level = Dir_branch.access_level;
  dir_sons_vol_id = Dir_branch.sons_vol_id;
  Master_Dir_switch = (type = "directory") &
     (dir_sons_vol_id ≠ sons_vol_id);

exception
  ^Inas(dir, "append");
  ^Dominates(access_level, dir_access_level);
  Entry_defined(dir_uid, entry);
  (type = "data") & ^Cur.mount_list(dir_sons_vol_id);
  Branch_offset(dir_uid, entry, type) > max_length;
  (NPFB(dir_uid, entry, type) &
     (Quota(dir).quota = Quota(dir).quota_used));
  (access_level ≠ dir_access_level) & (new_quota = 0);
  ^Master_Dir_switch  &
      ((NPFA(dir_uid, entry, type) &
          (Dir_branch.quota - Dir_branch.quota_used - 1 < new_quota)) |
       (^NPFA(dir_uid, entry, type) &
          (Dir_branch.quota - Dir_branch.quota_used - 0 < new_quota)));
  if Master_Dir_switch
     then new_quota = 0;
          ^LV_defined(sons_vol_id);
          Cur.access_level ≠ LVRF(sons_vol_id).access_level;
          LVRF(sons_vol_id).quota < new_quota;
  end

effect
  Directory(dir_uid, entry).uid = ´Unique_name;
  Directory(dir_uid, entry).type = type;
  Directory(dir_uid, entry).access_level = access_level;
  Directory(dir_uid, entry).ACL = "undefined";
  Directory(dir_uid, entry).last_acli = 0;
  Directory(dir_uid, entry).quota_given = new_quota;
  Directory(dir_uid, entry).quota = new_quota;
  Directory(dir_uid, entry).quota_used = 0;
  Directory(dir_uid, entry).TRP = 0;
  Directory(dir_uid, entry).LT = ´Current_calender_time;
```

Figure 24. Create_segment Function

```
caseof type
   "data"!
      [(¥ioffset)(Data(seg_uid, ioffset) = 0;)
      Directory(dir_uid, entry).vol_id = dir_sons_vol_id;]
   "directory"!
      [(¥ientry)(Directory(seg_uid, ientry) = "undefined";)
      Directory(dir_uid, entry).vol_id = root_vol_id;
      Directory(dir_uid, entry).sons_vol_id = sons_vol_id;]
end;
if Master_Dir_switch
   then LVRF(sons_vol_id).quota = *-new_quota;
   else Dir_branch.quota = *-new_quota;
end;
if ´NPFB(dir_uid, entry, type)
   then effects_of Update_quota(dir, +1);
        Page_allocated(dir_uid, branch_page) = "true";
end;
effects_of Audit("Create_segment", access_level);
```

Figure 24. Create_segment Function (concluded)

The basic security requirement for creating a segment is that the user have append access to the parent directory. Also, the access level of the new segment must dominate its parent's, to preserve the compatible hierarchy axiom.

A check must be made to ensure that entries are unique within a directory. If a data segment is being created, the volume it will be created on must be mounted. The remaining four exceptions are associated with the quota mechanism.

First, the new segment cannot be created if there is no space in the directory for the new branch. There are two possible forms of overflow: 1) segment overflow -- the segment offset where the kernel wants to store the branch is beyond the segment's maximum length; and 2) quota overflow -- storing a new branch would require a new page, and all of the directory's quota is used up. Second, if the new segment is an "upgraded" segment it must be given a terminal (non-zero) quota. Third, if the new segment is given a terminal quota, it must not be greater than the available quota in the cell from which it is taken. For segments other that master directories, the terminal quota is taken from the parent. And fourth, a master directory must be given its own quota, taken from the quota cell associated with the logical volume. Since this operation observes and modifies the logical volume state information, the user must be at the level of the volume.

The effect statements of Create_segment set the segment attributes in the branch to the values specified by the function arguments or to the standard initial values (zero words for data segments, undefined entries for directories). The contents of the new segment is set to the standard initial values, and quota is adjusted. In performing the quota manipulation there are two factors to consider: 1) the new segment is or is not a master directory; and 2) creating the new segment has or has not increased the size of the parent directory. Finally, the operation is audited.

The basic security requirement enforced by the Delete_segment function is that the user must have modify access to the parent directory (see Figure 25). In addition, the kernel does not permit upgraded directories or non-empty directories to be deleted. Allowing a non-empty directory to be deleted would add considerable complexity to the kernel, because the entire sub-tree rooted at the non-empty directory would have to be deleted, and this sub-tree could contain master directories. Since non-empty directories cannot be deleted, neither can upgraded directories, since the user of this function must be at the level of the parent directory and cannot be told anything about the contents of an upgraded offspring.

```
O_function Delete_segment(dir, entry)

let
  dir_uid = Cur.KST(dir).uid;
  seg_uid = 'Directory(dir_uid, entry).uid;
  quota = 'Directory(dir_uid, entry).quota_given;
  quota_used =
    cardinality({ipage | Page_allocated(seg_uid, ipage) = "true"});
  sons_vol_id = 'Directory(dir_uid, entry).sons_vol_id;
  Master_Dir_switch = 'Directory(dir_uid, entry).type = "directory" &
                                  sons_vol_id ≠ Dir_branch.sons_vol_id;

exception
  ^Inas(dir, "modify");
  ^Entry_defined(dir_uid, entry);
  (Directory(dir_uid, entry).type = "data") &
     ^Cur.mount_list(Dir_branch.sons_vol_id);
  (Dir_branch.access_level ≠ Directory(dir_uid, entry).access_level &
     (Directory(dir_uid, entry).type = "directory";
  if (Directory(dir_uid, entry).type = "directory")
     then ((∔ientry)(^Entry_defined(seg_uid, ientry)));
  end

effect
  Directory(dir_uid, entry) = "undefined";
  (¥iprocess_id, iseg)
    (if 'Process(iprocess_id).KST(iseg).uid = seg_uid
       then Process(iprocess_id).KST(iseg) = "undefined";
    end);
  (¥ ipage)(Page_allocated(seg_uid, ipage) = "false");
  if quota = 0
     then effects_of Update_quota(dir, quota_used);
  end;
  caseof 'Directory(dir_uid, entry).type
     "data"! (¥ioffset)(Data(seg_uid, ioffset) = "undefined");
     "directory"! (¥ientry)(Directory(seg_uid, ientry) = "undefined");
  end;
  if Master_Dir_switch
     then LVRF(sons_vol_id).quota = *+quota;
     else Dir_branch.quota = *+quota;
  end;
```

Figure 25. Delete_segment Function

The Delete_segment function has four distinct effects: 1) the segment's branch is set to undefined; 2) the segment is removed from all process address spaces; 3) the contents of the segment are set to undefined; and 4) all the pages allocated to the segment are released and the quota is recovered from the segment. If the segment did not have its own quota cell, the quota used by the released pages is returned via Update_quota. If the segment did have its own quota cell, the quota given to it is returned either to its parent or to the logical volume if the segment was a master directory.

Adding and Removing ACL Elements

The Add_ACL_element and Remove_ACL_element functions correspond to the model transition functions give access and rescind access (Figure 26). The correspondence is not exact because adding an element to a Multics ACL may rescind access rights (for an explanation, see Saltzer [14]).

The parameters of Add_ACL_element identify a directory in the user's address space (dir), a segment inferior to that directory (entry), a position in the segments ACL (acli), and an ACL element (principal_id, access_modes). The exceptions check the basic security requirement (that the user has "modify" access to the directory), that the segment exists, that acli is not beyond the current length of the ACL, and that adding the new element will not cause segment overflow or quota overflow.

The effect statements move all of the existing ACL elements from acli to the end of the ACL back one, insert the new element at acli, and increment the last_acli attribute by one. Also, if NPFA indicates that a new page is required, the quota is updated and the page is marked as being allocated.

Since the kernel adds new elements where the user indicates, rather than determining a position for them, the kernel does not guarantee any particular ordering of ACL elements with respect to "don't care" indicators in the three fields of the principal identifier. Nevertheless, since the "user" of the kernel is really the supervisor, it can maintain compatibility by inspecting the segment's current ACL and determining the proper position for a new element before calling Add_ACL_element.

The Remove_ACL_element function is simpler than Add_ACL_element because the kernel does not automatically free any pages that become unused. The user specifies a directory, a segment within the directory, an element on the segments ACL (by acli, the position of the element). Again, the security requirement is that the user have "modify" access to the directory. The exceptions also check that the

```
0_function Add_ACL_element(dir, entry, acli, principal_id, access_modes)

let
  dir_uid = Cur.KST(dir).uid;
  acli_page = Acle_offset(dir_uid, entry, principal_id)//page_size;

exception
  ^Inas(dir, "modify");
  ^Entry_defined(dir_uid, entry);
  acli > Directory(dir_uid, entry).last_acli + 1;
  Acle_offset(dir_uid, entry, principal_id) > max_length;
  (NPFA(dir_uid, entry, principal_id) &
     (Quota(dir).quota = Quota(dir).quota_used));

effect
  (¥iacli)
    (if (acli ≤ iacli ≤ 'Directory(dir_uid, entry).last_acli)
       then Directory(dir_uid, entry).ACL(iacli+1) =
              'Directory(dir_uid, entry).ACL(iacli);
    end);
  Directory(dir_uid, entry).ACL(acli).id= principal_id;
  Directory(dir_uid, entry).ACL(acli).mode= access_modes;
  Directory(dir_uid, entry).last_acli = *+1;
  if 'NPFA(dir_uid, entry, principal_id)
     then effects_of Update_quota(dir, +1)
          Page_allocated(dir_uid, acli_page) = "true";
  end;
```

Figure 26. Add_ACL_element and Remove_ACL_element Functions

```
O_function Remove_ACL_element(dir, entry, acli)

let
  dir_uid = Cur.KST(dir).uid;
  end_acli = 'Directory(dir_uid, entry).last_acli;

exception
  ^Inas(dir, "modify");
  ^Entry_defined(dir_uid, entry);
  acli > end_acli;

effect
  (¥iacli)
    (if (acli < iacli ≤ end_acli)
       then Directory(dir_uid, entry).ACL(iacli-1) =
              'Directory(dir_uid, entry).ACL(iacli);
    end);
  Directory(dir_uid, entry).ACL(end_acli) = "undefined";
  Directory(dir_uid, entry).last_acli = * -1;
```

Figure 26. Add_ACL_element and Remove_ACL_element Functions (concluded)

segment exists and that acli is not beyond the end of the ACL. The effect statements remove the element by moving all of the elements in the list behind it up one position, and decrementing last_acli by one. If a page of the directory does become unused, the user must explicitly invoke a kernel function to deallocate it.

Entering Messages

The remaining O-functions in the storage system module access segments as permitted by the security axioms -- they do not correspond to model transition functions because they do not change the security state.

The kernel supports the non-kernel supervisor's creation of a multi-level message facility by providing a function to enter a message into a higher level data segment, Enter_msg (Figure 27). Although the other operations on message segments (observing messages, removing messages, and entering messages at the user level) can be performed by the interpreter functions for reading and writing data segments, Enter_msg must be provided by the kernel, because a process entering a message into a higher level segment cannot observe the contents of the segment. To prevent existing messages from being over-written, an interpretive mechanism must be provided by kernel.

The kernel must be aware of the way data segments for messages will be implemented. The Message_offsets function (Figure 18) returns the set of offsets that the kernel may use for storing a new message in a data segment. If the segment is not in the proper format for messages, a null set is returned.

Enter_msg, like all other functions that store new information into segments, must recognize the possibility of segment and quota overflow -- the function cannot store a new message in a segment if there is no room for it. Unlike the other functions, however, this function cannot have an exception that checks for quota overflow, because it would allow the user to know about the segment contents. Therefore, the effect statements storing the message into the segment are conditioned upon there not being quota or segment overflow. If overflow does occur, the message will not be sent, but the user will not know about it. This loss of information cannot be avoided if multi-level communication is to be secure.

Quota Motion

The Move_quota function (Figure 28) moves a specified amount of quota from a directory to one of its offspring (which may also be a directory). If a negative quota is specified the actual movement of quota is in the opposite direction; an attempt to move zero quota

```
O_function Enter_msg(seg, msg_value)

let
  seg_uid = Cur.KST(seg).uid;
  msg_offsets = Message_offsets(seg_uid, msg_value);
  msg_pages = {msg_offsets//page_size};
  new_pages = {ipage | (ipage ∈ msg_pages) &
    (~'Page_allocated(seg_uid, ipage))};

exception
  ^Inas(seg, "enter");

effect
  if Quota(seg).quota > Quota(seg).quota_used + cardinality(new_pages)
     then (∀ioffset)
             (if ioffset ∈ msg_offsets
                 then Data(seg_uid, ioffset) = msg_value;
             end);
          if cardinality(new_pages) > 0
             then effects_of Update_quota(seg, +cardinality(new_pages));
                  (∀ ipage ∈ new_page)
                     (Page_allocated(seg_uid, ipage) = "true");
          end;
  end;
```

Figure 27. Enter_msg Function

```
O_function Move_quota(dir, entry, quota)

let
  dir_uid = Cur.KST(dir).uid;
  quota_used = Directory(dir_uid, entry).quota_used;
  seg_uid = Directory(dir_uid, entry).uid;
  upgraded =
    Dir_branch.access_level ≠ Directory(dir_uid, entry).access_level;

exception
  ^Inas(dir, "modify");
  ^Entry_defined(dir_uid, entry);
  quota = 0;
  ((Directory(dir_uid, entry).type = "directory") &
     (Directory(dir_uid, entry).sons_vol_id ≠ Dir_branch.sons_vol_id));
  Dir_branch.quota - quota < Dir_branch.quota_used;
  if upgraded
     then quota < 0;
     else Directory(dir_uid, entry).quota = 0;
          Directory(dir_uid, entry).quota + quota = 0;
          Directory(dir_uid, entry).quota + quota <
            Directory(dir_uid, entry).quota_used;
  end

effect
  Directory(dir_uid, entry).quota_given = *+quota;
  Directory(dir_uid, entry).quota = *+quota;
  Dir_branch.quota = *-quota;
```

Figure 28. Move_quota and Release_page Functions

```
O_function Release_page(seg, page)

let
  seg_uid = Cur.KST(seg).uid;

exception
  ^(Inas(seg, "write") | Inas(seg, "modify"));
  ^Page_allocated(seg_uid, page);
  if Branch.type = "directory"
     then Dir_page_exists(seg_uid, page);
  end

effect
  effects_of Update_quota(seg, -1);
  Page_allocated(seg_uid, page) = "false";
  if Branch.type = "data"
     then (¥ioffset)
            (if (0 ≤ ioffset) & (ioffset < page_size)
               then Data(seg_uid, page+ioffset) = 0;
            end);
  end;
```

Figure 28. Move_quota and Release_page Functions (concluded)

causes an exception. The basic security requirement is that the user must have "modify" access to the parent directory, because segment attributes are being observed and modified.

Quota cannot be moved between a directory and its children if they are on different volumes, since the physical implementation of volumes prevents realizing such a transfer. If quota is being moved from the parent directory, enough must be left to cover its present quota used.

Since the user is not entitled to know the quota used of an upgraded segment (it is information at a higher level), quota cannot be removed from an upgraded segment. For all other directories, quota motion cannot cause a change from terminal to non-terminal or visa versa (i.e., a change of quota to or from zero). Finally, if quota is taken away from a non-upgraded segment, enough must be left to cover the quota used. The effects of Move_quota simply make the update if all the exceptions are passed.

The last storage O-function, Release_page (also Figure 28), controls the deallocation of segment pages. Unlike allocation, which happens automatically when a page is first required to record non-zero information, the deallocation of pages will occur only on the explicit invocation of this function. Although this mechanism differs from the current Multics (which appears to be non-deterministic to the user), it introduces no functional incompatibilities and could be hidden by the interpreter. The motivation for this function is the need for all visible kernel effects to be deterministic for a proof of security.

The exceptions check that the user has some form of modify access to the segment specified and that the page to be released is presently allocated. If the segment is a directory and the page non-zero, the release is not allowed as it would damage kernel data. No restrictions apply to data segments.

The effect of Release_page is to deallocate the specified page and update the quota cell it was charged to. If the segment was a data segment the page is zeroed, nullifying any previously contained information.

STORAGE CONTROL REVIEW

Compared to the specifications of other subsystems, the storage system specification is rather large. This bulk is due to a number of factors, principally the complexity of specifying 1) the directory hierarchy; 2) segment sharing among all processes; 3) the access control list mechanism; and 4) the Multics quota mechanism. We

believe, however, that storage system is well understood and that it presents no conceptual difficulties (with respect to security correctness) because there is a close correspondence between it and the model.

SECTION V

PROCESS MANAGEMENT

The process management module of the kernel provides the security-related process functions: interprocess communication, process creation, and process deletion. A process is a program in execution on one of the virtual processors defined by the kernel [16].Normally, a process is created for a user when he logs in to the system and deleted when he logs out. Processes may be created in response to user "absentee" requests and driven by a data segment. There are also system ("daemon") processes, created at system startup, that exist indefinitely, performing specialized functions.

In Volume I, an introduction was given to the current Multics process organization, the design decisions that defined the kernel process system were reviewed, and the kernel process management functions were introduced.

With the background of Volume I, in this section, the formal specification of the kernel process management interface will be presented and the compatibility of the kernel with the existing system will be discussed.

CURRENT DESIGN -- DETAILS

The current Multics design contains two facilities that must be supported by the kernel process management module: process creation and deletion and interprocess communication. Volume I gave an overview of the requirements in each of these areas. This subsection details the functions to be compatibly supported by the kernel.

Process Creation and Deletion

Process creation initializes the set of process attributes. These include an entry in the active process table (APT), a process directory, a descriptor segment, a known segment table (KST), a set of registers, and various stack and temporary segments. In effect, a new address space and point of execution are defined.

In practice, the ability to create a process can be restricted to a small class of privileged processes, such as the answering service. Process deletion is a cooperative operation between the process to be deleted and its creator. A process cannot completely delete itself; another process must clean up the remains of the deleted process.

Interprocess Communication

The existing interprocess communication (IPC) mechanism consists of several layers. The top layer (user interface) provides operations on per-process communication paths called "event channels". The top layer of the interprocess communication mechanism is supported by a lower-level mechanism in the Multics traffic controller (TC).

The kernel will support the second layer of the IPC mechanism: the block/wakeup mechanism (not to be confused with the wait/notify mechanism, also provided by the traffic controller, supporting system/process communication). Wakeup causes the entry of a specified message in a global table of pending messages (interprocess transmission table -- ITT) and signals the specified process of this state. Block either returns the messages pending for the process that executes it or, if there are no messages pending, suspends the process and causes a ready process to start running. Ring 0 software, external to the TC, interprets messages to create the event channel system from the primitives. (These mechanisms are discussed in detail in [16].)

All other Multics process mechanisms (e.g., faults, interrupts) affect only process-local data and thus are not security issues. Process-local operations cannot cause any information flow outside the process. These operations can be supported by the interpreter mechanism.

THE KERNEL DESIGN

In Volume I, the kernel process design was introduced and the impact of the issue of user authentication was discussed. In this subsection, the implementation of the security and correctness requirements is discussed.

The kernel process management interface provides functions for creating and deleting processes. In this context the process being operated upon is an object, and the active process is a subject. A process may only create or delete a process at an equal or higher access level, because both actions involve a modification of information.

Because of the security controls provided by the kernel, the create and delete process functions can be provided to uncertified software. The security controls prevent processes from being used as Trojan Horses or from "spoofing" users at terminals. Spoofing is prevented because terminals become protected objects (of the secure

front end processor) once their access level has been set [17]. Like all other objects protected by the kernel, terminals are prevented from releasing or receiving information at the wrong levels.

The logging in of users and the creation of processes for them will thus be performed by the uncertified software constituting the Answering Service. In an absentee process the user's terminal is replaced by a pair of segments (one for input and one for output).

The kernel supports a block/wakeup mechanism similar to that provided by the Multics traffic controller. It maintains an interprocess transmission table of pending messages for each process and enforces security on the interprocess communications: the sending process is a subject altering an object, the receiving process.

Uncertified software in the supervisor will use the kernel-provided primitives to emulate event channels. The wait/notify mechanism is used internally to effect the virtual environment provided by the kernel. It is not used by uncertified software and is not supported at the kernel interface.

Process-local information and operations are not security issues. They will be supported by the uncertified software and hardware of the interpreter. The process-local data bases required to support the process local functions (e.g., registers, status bits, instruction counters) are supported by the interpreter module.

In summary, the kernel process management design provides an environment that will support the current user interface. The only restriction imposed is a result of the rigorous enforcement of security -- interprocess communication is restricted to be uni-directional between processes of different levels.

COMPATIBILITY WITH THE CURRENT MULTICS

The process management module addresses compatibility with Multics by presenting an interface that, in cooperation with uncertified software and hardware, will provide a compatible user interface. Processes are supported by maintaining data structures containing all process-relevant information as required by security.

Although the block/wakeup primitives provided by the kernel are simpler than those at the current ring zero interface, they are equivalent to the primitives available at the traffic controller (TC) interface. Since the current ring zero primitives are built on the TC primitives, the kernel provides a compatible interface. For instance, event-channel names can be added to messages by non-kernel software.

The majority of the answering service and absentee functionality will be implemented in uncertified code using kernel primitives such as Create_proc. For security, the initial determination of the terminal level is handled by certified software in the secure front end processor (SFEP). The Multics kernel will receive information about terminal activity via kernel-to-kernel communication with the SFEP.

Absentee processing can be controlled by absentee monitoring daemons that will receive requests for absentee service through the message system. These daemons will be created outside the kernel at each level at which absentee processing is to take place.

The kernel hardware primitives provided by the interpreter module are sufficiently general to support an emulation of the Series 60/level 68 hardware and all process-local effects.

SPECIFICATION

The process management specification is concerned with the process as an abstract entity. In addition to process creation, it describes the process environment and the security-related functions processes can execute.

Basic Definitions

The basic process definitions are concerned with the data structures that implement the process abstraction. These data structures contain all the security related information required by the kernel to implement virtual processes.

Processes

The principal type definition in process management is "process_type". The remaining definitions serve to more completely define "process_type". "process_type" is a structure containing all pertinent process information (see Figure 29). Many of the fields are self-explanatory. A few of note include: "principal_id", the identifier of the user on whose behalf the process is acting; "semaphore" and "notices", used to implement Block and Wakeup; "mount_list", the logical volumes mounted for the process; and "KST", the known-segment table of the process.

Known Segment Table

The known segment table (KST) defines the virtual address space of the processes. Each process managements its address space by moving entries in and out of the KST (using the Initiate and Terminate

```
  /* process_control type definitions*/

type

  notice_type = vector (0 to notice_length) of boolean

  process_type =
      structure(access_level: access_level_type
                principal_id: principal_id_type
                semaphore: integer
                notices: vector(0 to infinity) of notice_type
                mount_list: vector(0 to infinity) of uid_type
                KST: KST_type)

  kste_type =
    structure (dir: seg_type
               entry: entry_type
               uid, vol_id: uid_type)

  KST_type = vector(0 to max_seg_no) of kste_type
```

Figure 29. Process Control Type Definitions

functions of Section IV). Each KST entry (defined by "kste_type") records the parent of the segment, the segment's name and unique identifier, and the identifier of the volume it is mounted on. A process can only access segments that have been entered in its KST.

Process Control Parameters and Constants

The process management parameters and constants are given in Figure 30. The types referenced that do not appear in process management type definitions are defined in the common module. The constant "infinity" is the maximum value of the calendar clock, which, although finite, appears infinite in relation to the actual lifetime of any one system. It is used because the specification type definition mechanism makes it difficult to deal with infinite objects.

The constant "initial_KST" is a KST with entries for all the segments needed to start up a process: the kernel segments, temporary segments, and one procedure segment in the supervisor ring that will initialize the supervisor. This constant is used by the Create_proc function to circumvent the necessity of certifying a user-supplied KST. The implementation may require a compromise: a constant initial KST with some user-supplied entries.

The constant "initial_interpreter_data" is similarly used to initialize each process. It specifies, among other things, the starting address and initial register contents of the (virtual) processor. This information must be supplied by the kernel to ensure that each process starts in a secure state.

Hidden V-functions

The process management hidden V-functions are also given in Figure 30. They are both primitive V-functions. The hidden V-function Process embodies for each process all the information required by the kernel to support virtual processors. Process is in many respects similar to the Multics APT, "Process(process_id)" must be defined for the identifier of every active process; however, Process really represents all process security information. In an implementation, this information will be represented in various ways: global tables, temporary segments, and other information. Non-security process information will be maintained by the interpreter in a manner similar to directory images. This information may be thought of as part of the interpreter data or as residing in a segment accessible only to the process.

The hidden V-function Cur_process is the only indication given by the specification of the multiplexed processing capabilities of Multics. Its value is the identifier of the "current" process,

```
  /* process_control parameters*/

parameter

  initial_ga: ga_type

  nid,
  notice: notice_type

  pid: principal_id_type


  /* process_control constants */

constant

  infinity: integer = 2^time_length-1

  max_buffer_size: integer

  initial_KST: KST_type

  notice_length: integer

  initial_interpreter_data: vector(0 to *) of boolean

  max_notices:  integer


  /* process_control Hidden_V_functions */

Hidden_V_function Cur_process: uid_type
  /* the uid of the current process*/


Hidden_V_function Process(process_id):process_type
  /* all process information  */
```

Figure 30. Process Control Parameters, Constants, and Hidden V-functions

the process on whose behalf the kernel is executing. All kernel functions implicitly reference this function to determine the attributes of the subject executing them.

No multiplexing policy is implied by this function; one can imagine a distributed kernel simultaneously executing all processes. The derivation of this function is an implementation detail. Nevertheless, it must be such that its value is not modulatable or predictable. If it were, it could be an information channel.

The process management hidden V-functions are primitive functions and therefore must have an access level associated with them. The access level of Process is the access level of the process whose identifier is its argument. The access level of Cur_process is the access level of the process it identifies.

O-functions

Process control has no interface V-functions. The two process management functions that correspond to model transition functions are O-functions. There are also three O-functions supporting interprocess communication that observe or modify information.

Creating and Deleting Processes

Create_proc (Figure 31) is an OV-function. Its value is the identifier of the process created. It takes as arguments a principal identifier and an access level that are entered as attributes of the process.

The creation of a process modifies information at the level of the new process, so a check must be made to ensure that the creator (the current process) has alter access to that level. No other exceptions are checked. Although the usage of the Create_proc function must be carefully controlled by the supervisor for a compatible and functional Multics system, such control is not a security issue.

The effect of Create_proc is to initialize and set up a new entry in the Process V-function and make an entry recording the creation of the new process in the audit log.

Delete_proc (also Figure 31) is an O-function. It takes as an argument the identifier of the process to be deleted. Since processes are allowed to delete higher level processes, Delete_proc can have no exceptions. An exception that checked access to the process to be deleted would be an information path. (e.g., A message such as "process <high level> does not exist" reveals information at a level the user cannot have access to.

```
OV_function Create_proc(access_level, pid): uid_type

let
  process_id = ´Unique_name;
  createe = Process(process_id);

exception
  ^Secure_alter(Cur.access_level, access_level);

effect
  createe.access_level = access_level;
  createe.principal_id = pid;
  createe.semaphore = ∅;
  createe.KST = initial_KST;
  Interpreter_data(process_id) = initial_interpreter_data;
  effects_of Audit("Create_proc", access_level);

derivation
  process_id;



O_function Delete_proc(process_id)

effect
  if ´Process(process_id) ≠ "undefined" &
       Secure_alter(Cur.access_level, ´Process(process_id).access_level)
     then Process(process_id) = "undefined";
  end;
```

Figure 31. Create_proc and Delete_proc Functions

The "if" statement in the effects section of Delete_proc prevents the user from deleting any process he does not have correct access to. Again, the supervisor will restrict the use of delete process, most likely to self-deletion. The need for another process to clean up after a process deletes itself is an implementation effect that is not seen at the kernel interface.

Interprocess Communication

Block and Interrogate (Figure 32) are OV-functions. Block returns the oldest process notice. The age of the notices can be determined because the uid's are generated by the calendar clock. Block will cause a process to wait if there is no message pending. The effect "wait_until 'Cur.semaphore > 0" is intended to imply this wait;. the Block function will not complete if the semaphore would be non-positive (since there is no message to return). With respect to the process's own frame of reference, however, this delay is not seen.

If a message is pending, Block and Interrogate perform the same function, but if no message is pending, Interrogate returns immediately with an exception. In this way, a process can check for messages without incurring a wait.

Block and Interrogate read and alter information but only at the level of the current process. The semaphore and notices are information in "Process(Cur_process)"; therefore, no access checks are required.

Wakeup is an O-function. It sends a message to a process and notifies the process of the message. The effects of Wakeup are in the O-function macro Send_wakeup (Figure 33). The purpose of this separation is to provide a Multics kernel function that can be used by the SFEP kernel for communication. Since kernel-to-kernel communication occurs inside the kernel interface, the SFEP should not use the interface function Wakeup. A similar mechanism, SFEP_send_wakeup, is provided by the SFEP kernel for the Multics kernel (see Section VI).

Send_wakeup is designed to allow messages to be sent to processes at higher access levels. For the operation to be secure, it must not return any information about the process the message is sent to.

The operands of the "if" statement indicate that the message will not be sent if the process does not exist, the process message space is full, or the access is incorrect. (A message cannot be sent to a process with a lower access level.) These conditions cannot be treated as exceptions because they would reveal information about processes at higher levels. As with messages, if multi-level

```
OV_function Block: notice_type

let
  nid = min{inid | ´Cur.notices[inid] ≠ "undefined"};

effect
  wait_until ´Cur.semaphore > 0;
  Cur.semaphore = ´Cur.semaphore-1;
  Cur.notices[nid] = "undefined";

derivation
  ´Cur.notices[nid];



OV_function Interrogate: notice_type

let
  nid = min{inid | ´Cur.notices[inid] ≠ "undefined"};

exception
  ^´Cur.semaphore>Ø;

effect
  Cur.semaphore = ´Cur.semaphore-1;
  Cur.notices[nid] = "undefined";

derivation
  ´Cur.notices[nid];



O_function Wakeup(process_id, notice)

effect
  effects_of Send_wakeup(process_id, notice);
```

Figure 32. Block, Interrogate, and Wakeup Functions

```
O_function Set_principal_identifier(principal_id)

effect
  Cur.principal_id = principal_id;


  /* Process Control O-function Macros */

O_function_macro Send_wakeup(process_id, notice)

let
  wakee = Process(process_id);

effect
  if wakee ≠ "undefined" &
       'wakee.semaphore < max_notices &
       Secure_alter(Cur.access_level, wakee.access_level)
     then wakee.notices['Unique_name] = notice;
          wakee.semaphore = 'wakee.semaphore + 1;
  end;
```

Figure 33. Send_wakeup and Set_principal_id Functions

communication is to be secure, there will be the possibility of lost information.

If the message is allowed to be sent, it is entered in the "notice" buffer of the receiving process and the "semaphore" of that process is incremented. This operation can cause a pending Block to complete, if the process was waiting for a message.

Authentication Support

The O-function Set_principal_identifier (also Figure 33) and its use were described in detail in Volume I with regard to the issue of user authentication. Briefly, the use of the function must be controlled by the supervisor. The uncertified Answering Service will use it to set the identifier of each process it creates in response to a login. Because of the restrictions known to exist in the discretionary policy and because only process-local information is modified, providing this function to uncertified software is not a security issue.

PROCESS CONTROL REVIEW

The process management module describes the security-related functions required to support the process mechanism in Multics. The specification defines the access-control constraints that must be enforced on these functions for a valid interpretation of the security model. Although the facilities provided by the process management interface (process creation and deletion, interprocess communication, and support of the authentication mechanism) are only a subset of the existing Multics process functions, they are sufficient to allow an emulation of the current Multics process interface.

SECTION VI

EXTERNAL INPUT/OUTPUT

External I/O is the transmission of data between the internal secure computer environment and the external secure people/paper environment.[9] External I/O devices include terminals, card reader/punches, printers, and tapes. The external I/O primitives provided by the kernel permit the control of I/O operations by non-kernel software, subject to the appropriate security constraints.

In volume I, two varieties of external I/O were introduced -- communications I/O and peripheral I/O. The current design was not referenced because of its complexity and inherent insecurability. Volume I also introduced the concept of I/O coordinator (IOC) support, required because of the security implications of its operations.

In this section, the method used to circumvent the complexity of the present I/O scheme and to enforce security is explained. The implications of the method with regard to compatibility and implementation are also examined. With the discussions of Volume I as background, the formal specification of the kernel external I/O primitives are presented.

CURRENT DESIGN -- BACKGROUND

The current Multics I/O design consists of several layered modules that provide a sophisticated user interface. A device independent interface is provided for low speed devices such as terminals. For high speed devices, a more complex interface is provided, to allow device specific commands. Unfortunately, the existing I/O hardware did not contain sufficient functionality to support secure I/O.

In response to this problem, the front end processor that supported communications I/O has been replaced by a securable minicomputer (the SFEP) and hardware modifications will be made to the I/O multiplexer (IOM) that supports peripheral I/O to make it securable.

---

[9]Internal I/O supports the mapping of the one level virtual memory into the many levels of real memory (main memory, bulk store, secondary storage, etc.). Since internal I/O is performed entirely within the kernel, there is no notion of internal I/O at the kernel interface.

The SFEP will have its own kernel and the capability to securely control a large number of terminals or other low-speed devices. Communications I/O will be supported by kernel-to-kernel message transfer with the Multics kernel. The IOM hardware modification will confine every I/O path to an isolated domain, controllable by the Multics kernel. In particular, kernel controlled base and bounds registers will control every reference by a user I/O program to memory and the I/O start command that specifies the device the I/O program may access will be a kernel priviledged function.

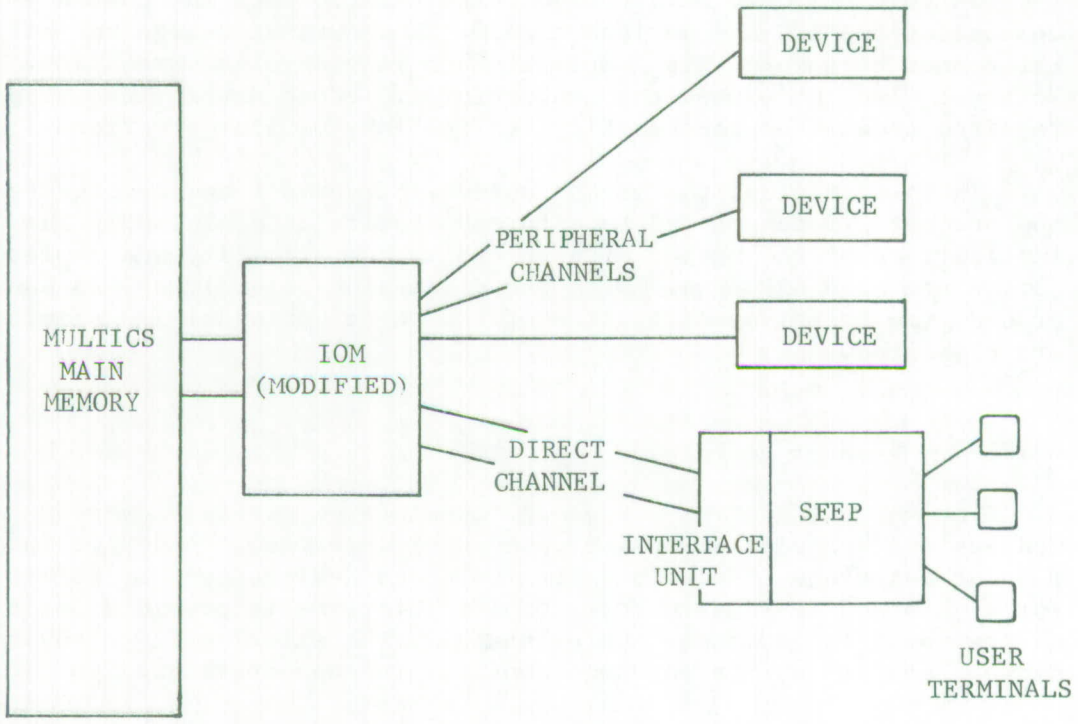

Figure 34. Kernel I/O Structure

Because these changes affect only the lowest level of I/O (never previously visible to the user), the user interface will remain the same. For the same reason, the software changes required for security will occur at such a low level that compatibility with the user interface is not an issue. The kernel provides a completely general low

level interface that is equivalent to the existing one (see Figure 34).

THE KERNEL DESIGN

Although it would be nice if all external I/O could be handled by the SFEP kernel with its sophisticated I/O mediation hardware, the requirements of high data rates and low level control for certain peripherals (e.g., tape drives, disk packs, card equipment) dictate two logical categories of external I/O -- communications I/O and peripheral I/O.

Communications I/O

Low-speed I/O, communications I/O, consists of terminals or terminal-like interfaces (e.g., a network interface). This type of I/O can be supported by the SFEP. Its kernel will handle the monitoring of terminals and will set up processes at the correct level to service them. Non-security functions such as code conversion for device independency can be handled by non-kernel SFEP software (for details, see [17].)

Because all terminals will be controlled by the verified SFEP kernel, the requirement for secure communications I/O is reduced to providing a kernel-to-kernel communication mechanism that will allow uncertified processes to communicate. The hardware interface will support a message type communication between the Multics and the SFEP kernels. The hardware will not enforce any security controls on this communication stream, since both sender and receiver will be verified kernel processes.

The software design provides a process-to-process communication path based on the process control block/wakeup mechanism. Since both kernels support this mechanism, communication is established by giving non-kernel software in one machine access to the Wakeup kernel function in the other machine. SFEP software will also be able to interrupt a Multics process (to transmit a quit signal) by setting a flag in the interpreter data of the process, but there is no need for a Multics process to interrupt the SFEP.

Figure 35 depicts the communications I/O path. The user process on Multics, the terminal process on the SFEP, and the terminal itself are all protected objects/subjects only allowed to communicate through the kernel. The kernels have a private interface to transfer the messages for all processes.

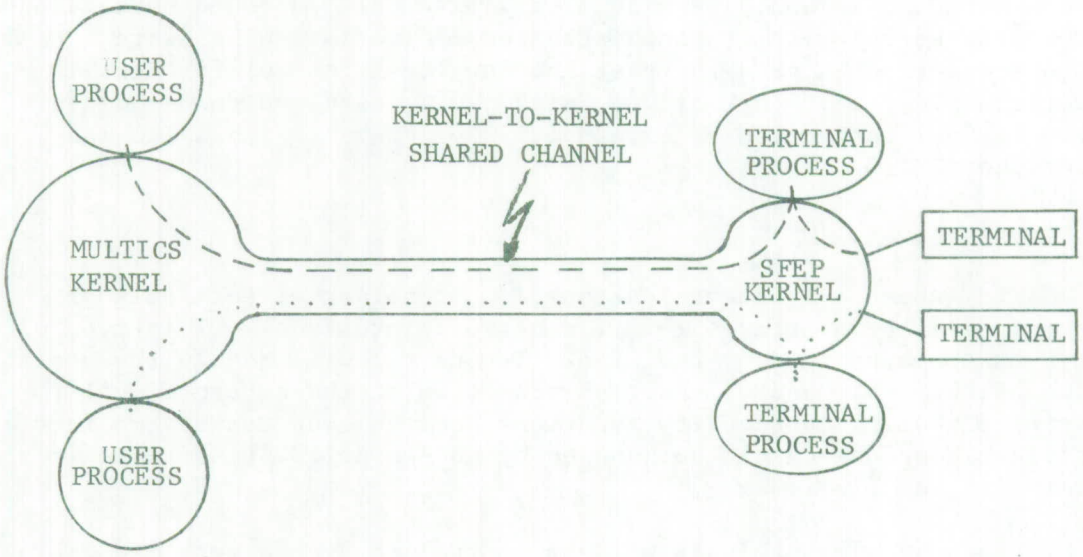

Figure 35. Communications I/O Paths

Peripheral I/O

High-speed or peripheral I/O will be supported by giving non-kernel software "direct"[10] access to peripheral devices connected to the system via the peripheral IOM (a separate IOM, inaccessible to users, will be used to provide internal I/O). This direct I/O will be made secure by hardware isolation. Only one user process will be allowed to perform I/O with each device at any one time. Each I/O path will

---

[10]The word "direct" is quoted here because although the user specifies his programs to execute directly on the IOM hardware, the startup of the program is done interpretively through the kernel. The kernel will use the priviledged start I/O command to set the area of memory and the device the user program is allowed to access.

be constrained to accessing a single area of memory and a single I/O device, both at equal access levels. Figure 36 conceptually illustrates this isolation.

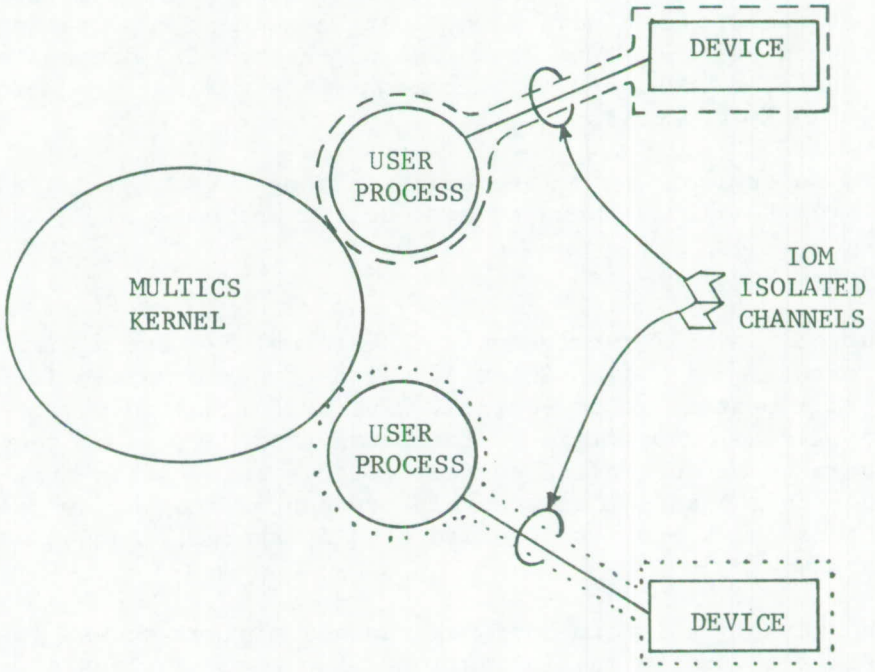

Figure 36. Peripheral I/O Paths

The modified IOM will have the hardware capability to provide isolation of each I/O path, thus preventing by default any possible breach of security. The Multics kernel will ensure that only one user controls each I/O path. There are several operational requirements that must be met to make the mechanism effective; they are discussed below.

All I/O events or signals must be fielded by the Multics kernel. It will know which process owns the signalling device from its device assignment table. It will signal that process through the block/wakeup mechanism or interpreter data system.

Input/Output Coordinator Support

The concept of IOC support was introduced in Volume I. The support provided is minimal, but will allow I/O daemons to securely service system_high, multi-level, or single-level line printers and card devices.

The kernel provides a function to copy a segment into a new directory. Presumably, this directory will be searched by an I/O daemon when it is ready to output. As with all upgrade functions, the copy can fail and the user cannot be notified of its failure. The chance of failure is minimized by doing a segment copy rather than using a word write-up function. The non-kernel I/O daemons and directories should be designed to accommodate the maximum probable load to avoid loss of information.

The supervisor can delete segments that have been copied to the daemon area to simulate the request delete option.

Operational Requirements

The operational requirements of external I/O are a direct result of its nature -- it is an interface with the external environment over which the computer has no control. The kernel has no way of determining the external properties of the peripheral devices or their media. For access to be controlled correctly, this information must be made available to the kernel and certain procedures must be established to ensure that the kernel is informed of all external changes as necessary.

The kernel is informed of maximum and minimum access levels of peripherals at system initialization. These access levels must agree with the physical environments of the devices. The actual access level of each device will be set by the system security officer on demand (see Volume III).

Terminals will be handled in a similar manner by the SFEP. Their control is more complex because of their close association with users. They will be used to establish user access levels and identities. The associated mechanisms go beyond the scope of this document and are detailed in [17]. For the Multics kernel, terminals are not a concern. They are simply another set of protected objects.

For peripherals with changeable media, there is an additional operational concern. The level of each medium must match the level of the device it is mounted on. The kernel has no way of verifying this property, so the operator must be suitably trusted.

COMPATIBILITY WITH THE CURRENT MULTICS

As was pointed out above, compatibility is not a major issue in the area of external I/O because all modifications occur at such a low level. The top level specification defines a completely general low-level I/O interface that is secure. The correspondence of each of the implemented interfaces to the specification will ensure their security.

The present I/O structure can be implemented on this low level interface with only minor changes. All non-security operations such as buffering will not be supported by the kernel and thus will move from their present ring, but with little change in functionality.

The SFEP can be thought of as simply a secure replacement for the previous front end. Its gross functionality is the same. The IOM interface will be the same except for the addition of security controls. The current I/O modules will run outside the kernel and thus be constrained by the partitioning of devices by level, but their functionality need not change.

SPECIFICATION

Basic Definitions

The basic definitions unique to the external I/O specification are shown in Figure 37. The "device_type" structure defines the information about external I/O peripheral devices that is relevant to the Multics kernel. This information includes maximum, current, and minimum access levels, the uid of the process to which the device is currently assigned, if any, and the "status" and "buffer" of the device. The current access level is the level at which the device is currently operating. It is set by a security reconfiguration operation (a system security officer function), and is always between the maximum and minimum levels, which are defined external to the kernel. "status" is device dependent; it includes such things as the current I/O command that the device is executing and error conditions (e.g., card reader hopper empty or parity error). "buffer" contains the data that the device is currently transmitting (e.g., a card image or disk record).

The hidden V-function Device (also Figure 37) defines the table of device information maintained by the kernel. Its access level is the current level of the device it describes.

```
  /* external I/O type definitions */

type

  device_type = structure
     (max_al,
     access_level,
     min_al: access_level_type
     owner: uid_type
     status: status_type
     buffer: buffer_type)

  status_type = vector (0 to status_size) of boolean

  buffer_type = vector (0 to buffer_size) of boolean


  /* external I/O parameters */

parameter

  buffer: buffer_type
  status: status_type
  to_dir_dir,
  to_dir_entry: seg_type


  /* external I/O constants */

  status_size: integer
  buffer_size: integer


  /* external I/O hidden_V_functions */

Hidden_V_function Device(device_id): device_type
  /*  all device information */
```

Figure 37. External I/O Basic Definitions

V-functions and O-functions

There are three distinct groups of external I/O functions. Communications I/O is secured by relying on the kernel in the front end processor. Peripheral I/O is secured by isolating each I/O path, an overly strict but easily implementable control. The IOC could be secured by forcing it to operate outside the kernel, but to maintain compatibility, a function is provided that will copy segments in the hierarchy under kernel control.

Communications O-functions

Figure 38 gives the specification of the Send function, used by Multics processes to send messages to the SFEP. It invokes the Send_wakeup kernel primitive provided by the SFEP kernel (this function is defined in [17]). The Multics Send_wakeup kernel primitive discussed in Section V is used by the SFEP to send messages back. With these functions, an uncertified Multics process can securely carry on I/O with the SFEP process controlling its terminal.

IOM O-functions

O-functions for operating external I/O devices attached to Multics via the IOM are shown in Figure 39. The Get_device function corresponds to the model's get access rule. This function allocates a device to a process, thus allowing the process to perform external I/O with the device. Since I/O inherently involves a bidirectional information flow between the process and the device -- at a minimum, the process must send commands to the device and receive back status information -- processes are only allocated devices at their own level. Also, simultaneous sharing of devices is prevented. This exception reflects the implementation choice made for the securing of peripheral I/O. The effect of Get_device is simply to record in the device table the identifier of the process to which the device has been allocated.

Release_device is the inverse of Get_device; it undoes the device allocation. Release_device corresponds to the model's release access function.

Once a process has been allocated a device, it can send and receive information until it releases the device. The process sends commands and data to the device with the O-function Write_device. Both the status and the buffer of the device may be modified. The exceptions check first that the process can know of the existence of the device and then that the process is the owner of the device.

```
  /*  SFEP communication O-function  */


O_function Send(proc, message)

exception
  (SFEP_Process(proc) = "undefined") |
     (SFEP_Process(proc).access_level ≠ Cur.access_level);

effect
  effects_of SFEP_Send_wakeup(proc, message);
```

Figure 38. Send Function

```
  /* IOM O-functions */

O_function Get_device(device_id)

exception
  ^Secure_write(Cur.access_level, Device(device_id).access_level);
  ^Device(device_id).owner = "undefined";
  device_id = SSO;

effect
  Device(device_id).owner = Cur_process;


O_function Release_device(device_id)

exception
  ^Secure_read(Cur.access_level,Device(device_id).access_level);
  ^Device(device_id).owner = Cur_process;

effect
  Device(device_id).owner = "undefined";


O_function Write_device(device_id, status, buffer)

exception
  ^Secure_read(Cur.access_level,Device(device_id).access_level);
  ^Device(device_id).owner = Cur_process;

effect
  Device(device_id).status = status;
  Device(device_id).buffer = buffer;



  /* external I/O V-functions */

V_function Read_device(device_id): structure(buffer: buffer_type,
                                             status: status_type)

exception
  ^Secure_read(Cur.access_level,Device(device_id).access_level);
  ^Device(device_id).owner = Cur_proc;

derivation
  buffer = Device(device_id).buffer;
  status = Device(device_id).status;
```

Figure 39. IOM Functions

A process receives data and status information from a device with the V-function Read_device (also Figure 39). Read_device has the same exceptions as Write_device and observes the status and buffer of the device. Note that the status information is static information. Status signals that cause interrupts are handled by the kernel as was described above.

IOM devices that actively access a process's address space (as a subject on behalf of a process) will be constrained by the modified IOM hardware to accessing a single segment. At the top level, such devices are envisioned as using the interpreter segment accessing functions. The actual hardware implementation will be more restricted, and thus a correspondence to prove security may be easily demonstrated.

IOC O-functions

The O-function Copy_segment (Figure 40) supports the IOC. It copies a data segment from one directory to another higher level directory by creating an upgraded segment with the same contents. The function has two pairs of arguments; the first two indicate the segment to be copied and the second two indicate the directory that will contain the segment copy.

The first two exceptions check that the user can observe the segment to be copied and that it is a data segment. The next two exceptions check that the user has specified a directory to copy to and that he can know of the directory's existence. The fourth and fifth exceptions check that the user really will be creating an upgraded segment (he cannot be allowed to copy the segment to a lower level). The last two exceptions ensure that the volumes involved are mounted.

The effects are conditional. They can only be completed if there is a free entry in the destination directory and there is sufficient quota to create the copy. If these conditions are passed, the effects are to create an upgraded segment whose contents are equal to the source segment. The effects are essentially the same as Create_segment.

EXTERNAL INPUT/OUTPUT REVIEW

The kernel design for external I/O offers no radical departures from the current Multics system. The kernel design consists of secure mechanisms for: 1) communicating with an SFEP, a secure replacement for the current, unsecurable Multics front end processor; 2) accessing external I/O devices attached to Multics via the modified IOM;

```
/* IOC O-function */


O_function Copy_segment(dir, entry, to_dir_dir, to_dir_entry)

let
  dir_uid = 'Cur.KST(dir).uid;
  seg_uid = 'Directory(dir_uid, entry).uid;
  to_dir_dir_uid = Cur.KST(to_dir_dir).uid;
  To_dir_branch = Directory(to_dir_dir_uid, to_dir_entry);
  to_access_level = To_dir_branch.access_level;
  seg_pages = cardinality{ipage|Page_allocated(seg_uid, ipage)};
  qc_dir_uid = QC_ptr(dir).qc_dir_uid;
  qc_entry = QC_ptr(dir).qc_entry;
  to_dir_uid = To_dir_branch.uid;
  delta_time = 'Current_calender_time - To_dir_branch.LT;
  delta_TRP = delta_time*'To_dir_branch.quota_used;
  new_uid = 'Unique_name;

exception
  ^Inas(dir, "status");
  Directory(dir_uid, entry).type ≠ "data";
  ^Inas(to_dir_dir, "status");
  To_dir_branch.type ≠ "directory";
  ^Dominates(to_access_level, Cur.access_level) |
     to_access_level = Cur.access_level;
  ^Dominates(to_access_level, Branch.access_level);
  ^Cur.mount_list(Dir_branch.sons_vol_id);
  ^Cur.mount_list(To_dir_branch.sons_vol_id);
```

Figure 40. Copy_segment Function

```
effect
  if ((∃ientry)('^Entry_exists(to_dir_uid, ientry)) &
     (if NPFB(to_dir_uid, ientry, "data")
      then To_dir_branch.quota - To_dir_branch.quota_used ≥ seg_pages + 1
      else To_dir_branch.quota - To_dir_branch.quota_used ≥ seg_pages)
     then Directory(to_dir_uid, ientry).uid = new_uid;
        Directory(to_dir_uid, ientry).access_level = to_access_level;
        Directory(to_dir_uid, ientry).type =
          'Directory(dir_uid, entry).type;
        Directory(to_dir_uid, ientry).ACL = "undefined";
        Directory(to_dir_uid, ientry).quota = 0;
        Directory(to_dir_uid, ientry).last_acli = 0;
        Directory(to_dir_uid, ientry).quota_used =
          'Directory(dir_uid, entry).quota_used;
        Directory(to_dir_uid, ientry).quota_given = 0;
        Directory(to_dir_uid, ientry).TRP =
          'Directory(dir_uid, entry).TRP;
        Directory(to_dir_uid, ientry).LT = 'Directory(dir_uid, entry).LT;
        if NPFB(to_dir_uid, ientry, "data")
        then To_dir_branch.quota_used = *+seg_pages+1;
        else To_dir_branch.quota_used = *+seg_pages;
        To_dir_branch.TRP = *+delta_TRP;
        To_dir_branch.LT = 'Current_calender_time;
        (∀ ipage)(Page_allocated(new_uid, ipage) =
          'Page_allocated(seg_uid, ipage));
        (∀ ioffset)(Data(new_uid, ioffset) = 'Data(seg_uid, ioffset);
  end;
```

Figure 40. Copy_segment Function (concluded)

and 3) sending service requests to I/O daemons. All three of these mechanisms, without (DoD) security controls, are present in the current Multics.

SECTION VII

SUMMARY

The formal specification of a security kernel for the Multics system has been given. This specification is sufficiently detailed to allow a verification of its security and allow a decision to be made about its ability to efficiently and compatibly support the current Multics user interface.

The top level specification supports further work on the Multics kernel in two directions. First, the specification is the basis for the validation of the kernel. Second, the top level specification will guide the design and specification of the lower levels of the kernel. The lower level specifications will identify the mechanisms to be used to implement the security kernel.

The top level specification is completed by the specification of initialization, reconfiguration, and the trusted subject interface. These subsystems are detailed in Volume III.

APPENDIX I

INDEX TO SPECIFICATIONS

Functions

ACL_type 46
Access permission 52
Acle_apply 52
Acle_offset 55
Add_ACL_element 70
Audit 23
Audit_log 22
Block 88
Branch 20
Branch_offset 55
Copy_segment 103
Create_proc 86
Create_segment 65
Cur 20
Cur_process 84
Current_calendar_time 23
Data 49
Delete_proc 86
Delete_segment 68
Demount 61
Device 98
Dir_branch 20
Dir_page_exists 55
Directory 49
Dominates 22
Drive 49
Enter_msg 73
Entry_defined 56
Execute 30
Get_device 101
Inas 51
Initiate 62
Interpreter_data 29
Interrogate 88
KST_type 82
LVRF 49
LV_defined 56
Message_offsets 55
Mount 61
Move_quota 74
NPFA 54
NPFB 54
Page_allocated 49
Process 84
QC_ptr 54
Quota 54
Read 30
Read_audit_log 23
Read_device 101
Read_interpreter_data 30
Release_device 101
Release_page 75
Remove_ACL_element 71
Revoke_access 63
Secure_alter 22
Secure_read 22
Secure_write 22
Seg_attributes 58
Seg_side_effect_attributes 58
Send 100
Send_wakeup 89
Set_principal_identifier 89
Terminate 63
Test_and_set 32
Unique_name 22
Update_quota 59
Wakeup 88
Write 31
Write_device 101
Write_interpreter_data 31

Basic Definitions

access_level_type 17
access_mode 48
access_mode_type 46
access_modes 48
acli 48
acli_type 46
audit_log_type 17
base_seg 48
branch_type 46
buffer 98
buffer_type 98
calendar_time_type 17
category_type 17
char_string 17
class_type 17
compartment_type 17
device_id 19
device_type 98
dir 19
dir_level 19
dir_uid 48
drive_no 19
entry 19
entry_type 17
ga_type 17
iacli 48
idrive_no 48
ientry 48
il_type 17
initial_ga 84
interpreter_data 29
interpreter_data_type 29
ioffset 48
iprocess_id 48
iseg 48
itime 48
iuid 48
kste_type 82
level 19
levela 19
levelb 19
machine_word 48
machine_word_type 17
message_type 46
msg_value 48
new_entry 48
new_quota 48
nid 84
notice 84
notice_type 82
object_level 19
offset 19
offset_type 17
pid 84
principal_id 19
principal_id_type 17
process_id 19
process_type 82
quota 19
quota_change 48
ru_change 48
seg 19
seg_level 19
seg_type 17
seg_type_type 46
seg_uid 48
sl_type 17
sons_vol_id 48
status 98
status_type 98
subject_level 19
term_seg 48
test 29
time 19
to_dir_dir 98
to_dir_entry 98
type 48
uid_type 17
vol_id 48

APPENDIX II

SPECIFICATION LANGUAGE SYNTAX

The syntax is expressed in extended BNF with "+" and "*" used as they are in the construction of regular sets. Underlining is used to indicate terminal symbols that might otherwise be confused with symbols of the meta-language. The terminal character strings of the language are <number> (any string of numeric characters) and <symbol> (any string of characters, not all numeric).

```
<module>::=
          module <module_name>;
          <type>*
          <define>*
          <parameter>*
          <constants>*
          <Hidden_V_function>*
          <V_function>*
          <O_function>*
          <OV_function>*
          <V_function_macro>*
          <O_function_macro>*

<module_name>::=
          <symbol>

<type>::=
          type {<type_name>+ = <type_designator>;}+

<type_name>::=
          <symbol>

<type_designator>::=
          <simple_type>
          | <constructed_type>

<simple_type>::=
          <scalar_type>
          | <subrange_type>
          | <type_name>

<scalar_type>::=
          scalar(<constant>+)

<constant>::=
          "<symbol>"
```

```
          | <number>

<subrange_type>::=
          <simple_type>(<constant> to <constant>)

<constructed_type>::=
          <vector_type>
          | <structure_type>
          | <set_type>

<vector_type>::=
          vector <subrange_type> of <simple_type>

<structure_type>::=
          structure({<field_name>+ : <simple_type>}+)

<field_name>::=
          <symbol>

<set_type>::=
          set(<simple_type>)

<define>::=
          define {<identifier> = <expression>;}+

<identifier>::=
          <symbol>

<parameter>::=
          parameter {<fp_name>+ : <type_designator>;}+

<fp_name>::=
          <symbol>

<constants>::=
          constant {<identifier>+ : <type_designator>[ = <expression>];}+

<expression>::=
          <constant>
          | <identifier>
          | <function_call>
          | '<function_call>
          | <any reasonable mathematical operation on expressions>

<function_call>::=
          <fn_name>[(<expression>+)]

<fn_name>::=
          <symbol>
```

```
<Hidden_V_function>::=
          Hidden_V_function <fn_name>[(<fp_name>+)] : <type_designator>

<V_function>::=
          V_function <fn_name>[(<fp_name>+)] : <type_designator>
          [let <let>+]
          [exception <exception>+]
          [derivation <value>]

<let>::=
          <identifier> = <expression>;

<exception>::=
          <expression>;

<value>::=
          <expression>;

<O_function>::=
          O_function <fn_name>[(<fp_name>+)]
          [let <let>+]
          [exception <exception>+]
          [effect <effect>+]

<effect>::=
          <expression>;

<OV_function>::=
          OV_function <fn_name>[(<fp_name>+)] : <type_designator>
          [let <let>+]
          [exception <exception>+]
          [effect <effect>+]
          [derivation <value>]

<V_function_macro>::=
          V_function_macro <fn_name>[(<fp_name>+)] : <type_designator>
          [let <let>+]
          [derivation <value>]

<O_function_macro>::=
          O_function_macro <fn_name>[(<fp_name>+)]
          [let <let>+]
          [effect <effect>+]
```

REFERENCES

1. J.P. Anderson, "Computer Security Technology Planning Study," ESD-TR-73-51, Volume I, James P. Anderson & Co., Fort Washington, Pennsylvania, October 1972 (AD 758206).

2. R.R. Schell, P.J. Downey, and G.J. Popek, "Preliminary Notes on the Design of Secure Military Computer Systems," MCI-73-1, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, January 1973.

3. W.L. Schiller, "The Design and Abstract Specification of a Multics Security Kernel," ESD-TR-77-259, Vol. I, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., November 1977.

4. D.L. Parnas, "A Technique for Software Module Specification with Examples," Communications of the ACM, Volume 15, Number 5, May 1972, pp. 330-336.

5. W.R. Price, "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1973.

6. N. Wirth, "The Programming Language PASCAL," Acta Informatica, Volume 1, 1971, pp.35-63.

7. M.D. Schroeder and J.H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," Communications of the ACM, Volume 15, Number 3, March 1972, pp. 157-170.

8. C. Engelman, "Audit and Surveillance of Multi-level Computing Systems," ESD-TR-76-369, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., April 1977 (AD A039060).

9. W.L. Schiller, "The Design and Specification of a Security Kernel for the PDP-11/45," ESD-TR-75-69, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., May 1975 (AD A01171).

10. R.G. Bratt, "Minimal Protected Naming Facilities for a Computer Utility," MAC-TR-156, MIT Project MAC, Cambridge, Massachusetts, September 1975.

11. Multics Programmers' Manual, AG91 revision 1, AG92C revision 1, AG93C revision 1, and AK92 revision1, Honeywell Information Systems Inc., July 1976.

12. D.E. Bell and L.J. LaPadula, "Computer Security Model: Unified Exposition and Multics Interpretation," ESD-TR-75-306, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., March 1976 (AD A023588).

13. J.C. Whitmore, A. Bensoussan, P.A. Green, A.M. Kobziar, and J.A. Stern, "Design for Multics Security Enhancements," ESD-TR-74-176, Honeywell Information Systems, 1974.

14. J.H. Saltzer, "Protection and the Control of Information in Multics," Communications of the ACM, Volume 17, Number 7, July 1974, pp. 388-402.

15. D.K. Kallman, S.R. Ames, and S.M. Goheen, "Multics Security Kernel Validation," MTR-3384, Volumes I and II, The MITRE' Corporation, Bedford, Massachusetts (MTR in progress).

16. J.H. Saltzer, "Traffic Control in a Multiplexed Computer System," MAC-TR-30 (Thesis), MIT Project MAC, Cambridge, Massachusetts, July 1966.

17. M. Gasser, "The Top Level Specification of a Security Kernel for the Multics Front End Processor," ESD-TR-77-258, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., November 1977 (AD A047309).