INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in “sectioning” the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
McCaulley, Edwin John, III, 1947-
A Model for Data Secure Systems.

The Ohio State University, Ph.D., 1975
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48106
A MODEL FOR DATA SECURE SYSTEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

by

Edwin John McCauley III, B.S.E., M.S.E.

* * * * *

The Ohio State University

1975

Reading Committee:
Professor K.J. Breeding
Professor H.H. Buttelmann
Professor T.G. Delutis
Professor D.K. Hsiao
Professor F.A. Stahl

Approved by

Department of Computer and Information Science
PREFACE AND ACKNOWLEDGEMENTS

This document is the result of one research effort in a multi-faceted project on Data Secure Systems. The project was formed in the fall of 1972 and was funded in March, 1973. Dr. David K. Hsiao, Associate Professor of Computer and Information Science, is the principal investigator. Dr. D.S. Kerr, Associate Professor of Computer and Information Science, Dr. H.S. Koch, Assistant Professor of Computer and Information Science, and Dr. M.T. Liu, Associate Professor of Computer and Information Science are investigators on the project. There are nine Graduate Assistants on the project, R. Baum, D. Cohen, W. Horger, R. Hartson, N. Kaffen, R. Knablein, E. McCauley, C. J. Hee and D. Wong. Ms. Sandy Rich is the project secretary.

Six major research and experimentation efforts are underway. These are:

(1) Models for Data Secure Systems.
(2) Context Protection and Consistent Control in Data Secure Systems.
(3) Data Secure Computer Architecture.
(4) Design and Certification of Data Secure Systems.
(5) A System for Experimentation with Access Control Mechanisms.

(6) Languages and Language Translators for Protection Specifications.

It has been our policy to issue preliminary technical reports at significant milestones in the various projects. These will be referenced in the text of this document.

The support of the Office of Naval Research through contracts N00014-67-A-0232-0022 and N00014-75-C-0573 is gratefully acknowledged. Dr. David K. Hsiao, my advisor, has placed me in his debt through his tireless efforts during the course of this work. This debt is one which mere words cannot repay. I am also indebted to the members of my reading committee, Dr. K. Breeding, Dr. H.W. Buttelmann, Dr. T.G. Delutis, and Dr. F.A. Stahl for their significant contributions towards the polish of the final document. All of the members of the Data Security project have contributed to these results through their penetrating (occasionally too penetrating) analysis of the concepts. I would like to thank Mr. Frank Manola of the Naval Research Laboratory for the many stimulating discussions which we had. Their results may be found throughout this document. Ms. Sandy Rich, our secretary, deserves sincere thanks for putting up
with three demanding masters, Dr. Hsiao, me, and the most demanding, the computer on which this text was prepared. Her good humor in the struggles with the source data entry made the entire process easier. It is difficult to properly thank my wife, Sharon, for all that she did in course of the work. Her keen insight and excellent perspective helped immeasurably to keep me on course. Finally, I would like to thank my father and Mr. Richard Stofflett for introducing a small boy to an IBM-650 computer, thereby setting in motion the forces which resulted in this dissertation.
VITA

December 2, 1947

Born — Detroit, Michigan

1969

B.S.E. (Science Engineering) cum laude, The University of Michigan, Ann Arbor, Michigan

1970

M.S.E. (Computer, Information and Control Engineering), The University of Michigan, Ann Arbor, Michigan

1970-1972

Technical Intelligence Officer, U.S. Air Force, Hq. Foreign Technology Division (AFSC), Wright-Patterson AFB, Ohio

1973-1975

Research Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio
FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Computer Programming including Systems Programming. Professor David K. Hsiao

Studies in Computer Architecture. Professor F.A. Stahl

Studies in Automata Theory. Professor H.W. Buttelmann
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFACE AND ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>VITA</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>A NOTE ON THE TYPOGRAPHY</td>
<td>xii</td>
</tr>
<tr>
<td>Chapter I. INTRODUCTION AND OVERVIEW</td>
<td>1</td>
</tr>
<tr>
<td>Chapter II. THE CONCEPTUAL MODEL</td>
<td>9</td>
</tr>
<tr>
<td>Basic Definitions</td>
<td></td>
</tr>
<tr>
<td>Modelling Existing Protection Mechanisms</td>
<td></td>
</tr>
<tr>
<td>Four Types of Protection Specifications</td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td></td>
</tr>
<tr>
<td>Chapter III. THE STRUCTURAL MODEL</td>
<td>45</td>
</tr>
<tr>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>The Data Base Model</td>
<td></td>
</tr>
<tr>
<td>The System Model</td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td></td>
</tr>
<tr>
<td>Chapter IV. THE ENGINEERING MODEL</td>
<td>90</td>
</tr>
<tr>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>The Model</td>
<td></td>
</tr>
<tr>
<td>Extensions of the Basic Algorithms</td>
<td></td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td></td>
</tr>
</tbody>
</table>

viii
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 - System threats and countermeasures</td>
<td>3</td>
</tr>
<tr>
<td>1.2 - Computer systems penetration and protection</td>
<td>4</td>
</tr>
<tr>
<td>II.1 - Portion of an access matrix</td>
<td>20</td>
</tr>
<tr>
<td>II.2 - Access matrix expressed as a protection pattern</td>
<td>21</td>
</tr>
<tr>
<td>II.3 - Access matrix expressed as capability lists</td>
<td>23</td>
</tr>
<tr>
<td>II.4 - Records in a file specified by three logical descriptions</td>
<td>27</td>
</tr>
<tr>
<td>II.5 - Expression validations in record access</td>
<td>28</td>
</tr>
<tr>
<td>III.1 - Audited access</td>
<td>62</td>
</tr>
<tr>
<td>III.2 - Pass through problem</td>
<td>63</td>
</tr>
<tr>
<td>III.3 - A traffic deadlock</td>
<td>75</td>
</tr>
<tr>
<td>III.4 - Deadlock rule enforcement, before violation</td>
<td>77</td>
</tr>
<tr>
<td>III.5 - Deadlock rule enforcement, after violation</td>
<td>77</td>
</tr>
<tr>
<td>IV.1 - Records for atom example</td>
<td>97</td>
</tr>
<tr>
<td>IV.2 - Security atoms</td>
<td>98</td>
</tr>
<tr>
<td>IV.3 - Access types for entire file</td>
<td>119</td>
</tr>
<tr>
<td>IV.4 - Access types for subfiles</td>
<td>120</td>
</tr>
</tbody>
</table>
IV.5 - A possible protection record ................................................. 123
IV.6 - Improved protection record ................................................ 125
IV.7 - Records to be inserted ........................................................ 127
IV.8 - Potential security atoms ..................................................... 129
IV.9 - Records of class 2 ............................................................... 130
IV.10 - Data base with one record of class 2 ................................. 130
IV.11 - Record insertion sequence ............................................... 131
IV.12 - Resulting file ..................................................................... 133
IV.13 - Directory of file .................................................................. 134
IV.14 - File and access type structure for example .......................... 137
IV.15 - File Structure ..................................................................... 144

V.1 - BDSS Protection failure modes ............................................. 164

A.1 - BDSS module structure ......................................................... 189
A NOTE ON THE TYPOGRAPHY

This document was prepared on a DECsystem-10 computer using a version of the program RUNOFF modified by Mr. Serge Fournier. Due to limitations on the output devices, several conventions have been observed. Where there would be no confusion, subscripts have been "absorbed" into the basic symbol. For example, $K_t$ can be interpreted as "$K_{sub\ 1}$". If this was not suitable, the subscript was enclosed in square brackets, as would be the case in ALGOL. Thus, $\text{THETA}[i]$ is the $i$-th element of the array THETA. Certain special symbols were either replaced by their verbal description or were made from combinations of characters. Underlining is used to indicate that the term is being defined or for added emphasis.
Chapter I. Introduction and overview.

The topic of this dissertation is protection and access control, specifically protection and access control in highly secure data base management systems. It is perhaps indicative of a maturing technology that one would concern himself with protection in data base systems. Formerly, the problems of implementing a large scale data base system were so nearly insurmountable that protection was seldom even considered. Now, with the continued growth in the number and importance of data base systems, legitimate concern is being raised over the lack of security in many contemporary systems. Recent events have pointed out the alarming potential for harm that unscrupulous individuals using sophisticated technology possess.

At the outset the reader should be aware of where this work fits in the total picture of security. We are considering only a small portion of the total question. As Figures 1.1 and 1.2 (Figure 1.2 is based upon work originally created by the Rand Corp. and reproduced in [BearC72]) show, there are a wide range of options open to those who wish to attempt to penetrate a typical computer.
system. It has been stated that protection is like a chain, in that it is only as strong as the weakest link. Continuing the analogy, what we are attempting to do is to suggest mechanisms for strengthening an obviously weak link, that of the data base management system protection mechanism. To provide a Data Secure System, all of the links of the chain must be strengthened. One must accept that no system can be completely secure. If a penetrator has enough resources and determination, any system can be subverted. What the Data Secure System seeks to achieve is to raise the cost of "breaking" the system far above the potential rewards for doing so. Finally, one must realize that however perfect the protection mechanism may be, ultimately it must rely upon the inputs from fallible human beings. If an individual betrays the trust placed in him, either by accident or by design, the most elaborate access control system devised may not be able to circumvent the disclosure or modification of the protected data.
<table>
<thead>
<tr>
<th>Threat</th>
<th>Countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Operator Error</td>
<td>Production and Machine Controls</td>
</tr>
<tr>
<td></td>
<td>Separation of Duties</td>
</tr>
<tr>
<td></td>
<td>Quality Control Techniques</td>
</tr>
<tr>
<td>Input Errors</td>
<td>Program Checks</td>
</tr>
<tr>
<td></td>
<td>Control of Program Changes</td>
</tr>
<tr>
<td></td>
<td>Rigorous Testing</td>
</tr>
<tr>
<td>Programming Errors</td>
<td>Testing</td>
</tr>
<tr>
<td></td>
<td>Control of Program Changes</td>
</tr>
<tr>
<td></td>
<td>Formal Proofs</td>
</tr>
<tr>
<td></td>
<td>Internal Audits</td>
</tr>
<tr>
<td></td>
<td>Defensive Programming</td>
</tr>
<tr>
<td>Disaster</td>
<td>Backup Capability</td>
</tr>
<tr>
<td></td>
<td>Insurance</td>
</tr>
<tr>
<td>Hardware Failure</td>
<td>Production and Machine Controls</td>
</tr>
<tr>
<td></td>
<td>Defensive Programming</td>
</tr>
<tr>
<td>Sophisticated Interlopers</td>
<td>all of the above</td>
</tr>
<tr>
<td></td>
<td>countermeasures</td>
</tr>
<tr>
<td></td>
<td>Privacy Transformations</td>
</tr>
<tr>
<td></td>
<td>Threat Monitoring</td>
</tr>
</tbody>
</table>

Figure I.1 - System threats and countermeasures.
Figure 1.2 - Computer system penetration and protection.

Given this discussion of the context of the work, let us now outline our approach. Because of the diverse nature of the problem, it is quite difficult to propose a single, monolithic model. Rather, we have adopted the approach advocated by Zurcher and Randell [ZurcF68] of utilizing a series of models, each of which allows us to deal with a specific issue at a level of abstraction appropriate to the issue. One of the best examples of this approach is the
Data Independent Accessing Model of M.E. Senko et al [AltM72, AstrM72, SenkM72, SenkM73]. Our work was strongly influenced by their success in treating an issue of similar complexity. We have broken the problem down into four models.

On the highest level, which is called the **conceptual model**, we are concerned with a thorough description of the terms and concepts of protection. The notion of a protection pattern is developed. Intuitively, a protection pattern is a summarization of the result of every possible access attempt, i.e. whether the attempt would be permitted or denied. The protection pattern can be used to put all deterministic protection mechanisms into the same model. Since, however, the protection pattern is an unwieldy object, we suggest that the global protection pattern is made from distinct, local statements of protection requirements, which are termed protection specifications. By introducing the specifications, we can focus on the rules of the system (the specifications) rather than on their eventual mechanization (the protection pattern). This emphasis on policies rather than their mechanization is an important departure from previous work.

In the second level, the **structural model**, we are chiefly concerned with the structure of the data base and
the overall system concept. A general data base model is developed which incorporates most contemporary file structures as special cases. This model emphasizes the relations between records and utilizes these relations to provide and enhance protection. The resulting file structures are shown not only to have the necessary security considerations, but also to possess desirable characteristics in the areas of retrieval precision and system efficiency. The data base model is a major advance over previous work on the attribute based model in that we unify three diverse aspects of file design in a single, concise model.

The systems concept of the structural model is based upon the idea of multiprogramming deadlocks. Using this concept, an exact characterization of the necessary and sufficient conditions for preventing the completion of illegal accesses is developed. A variety of contemporary protection mechanisms are shown to be simply different mechanizations of this concept.

The third level of the model, termed the engineering model, is an effort to relate the largely theoretical results of the higher level models to the more practical considerations of typical data base management systems. Detailed algorithms are developed for the maintenance and
use of the data base. Both algorithms presented are new to this work.

Finally, in the lowest level we discuss a small system which was actually implemented. Due to the nature of this system and to the existence of other ongoing experiments with a more complete system, this document does not include very much experimentation with the small system. However, a detailed discussion of the correctness and completeness of the system is given. The techniques used are similar to the formal proofs of algorithms, but are somewhat more practical and less rigorous.

It is hard to discuss the major contributions of the work until we have developed some formalism. However, let us briefly outline the most significant aspects of this dissertation. The major contributions of this dissertation lie in three main areas. First, in the conceptual model we develop some new formalism for discussing protection. This formalism is based upon other work, but extends these concepts in a new and meaningful way. Second, the data base model is a significant advance over previous work. In a single unified treatment we relate three diverse aspects of logical file design. Finally, by suggesting that the data base management system design be "driven" by the protection requirements, we are able to make an important contribution
As a concluding remark, this document and the others which have employed the multi-level modelling technique attest to the success of the approach. With such a disciplined attack, difficult problems can be solved in a systematic way. In our case, we can carry the research from its most basic foundations to an actual implementation all within the same model.
Chapter II. The Conceptual Model

Basic Definitions

One of the problems that plague us in attempts to study protection is the lack of a suitable theoretical framework to allow for the comparison and analysis of the many different ideas for providing protection for database systems. The goal in building the conceptual model is to provide such a framework and to demonstrate its utility.

We shall try to use a more formal approach with the hope of arriving at more precise definitions. Nevertheless, there are things which must remain intuitive and somewhat "undefined".

Defn: The Physical Data Base (PDB) is the underlying physical reality of a database, e.g., a reel of magnetic tape, a deck of cards, a collection of disc tracks, etc.

Defn: The Logical Data Base (LDB) is the set of all elements of information contained in the PDB. Furthermore, the elements of information in LDB
are referred to as **logical data**.

**Defn.** A **Data Base System(s)** is (are) the collection of computer programs, procedures, and components etc. for the creation, use and maintenance of the Logical Data Base on the Physical Data Base.

Informally, the Logical Data Base is the collection of all possible "answers" obtainable or extractable by the Data Base System from the Physical Data Base in response to "questions" by users of the Data Base System.

**Assumption:** It is possible to enumerate completely every piece of logical data contained in the Physical Data Base.

This assumption often goes unstated since it appears so obvious. If we accept the assumption, however, it is not clear for many Data Base Systems how to form the "answers" since enumeration is hardly an efficient way to extract them. Nevertheless, the basis for this assumption is that if an element of data is capable of being "found" in the Physical Data Base by the Data Base System, then it can be enumerated.

The Logical Data Base will occupy the central position in any discussion of protection. Many Data Base System protection mechanisms have the ability to protect only the Physical Data Base, rather than the Logical Data Base. The
lack of more suitable protection makes many existing protection capabilities inadequate for modern, multi-user, integrated Data Base Systems. This inadequacy will be greatly multiplied since, as the current research in Data Base Systems suggests, there will be in the future even greater difference and separation between the Physical Data Base and the user's view of it, the Logical Data Base.

Now we define more formally some of the terms which have been frequently used in Data Base System technology.  

Defn: A user will be the generic term for any agent which attempts to use the Logical Data Base in some way.

Defn: Access is any activity by a user which requires logical data from the Logical Data Base for its completion.

Defn: An access is denied if it is indefinitely delayed (never completed); otherwise, it is permitted.

Access may be further subdivided into access types, such as read, write, execute, search, retrieve, etc. We shall leave the exact connotations of any particular access type unspecified at this point.

Let us now define some of the terminology more leisurely. The interaction of a user with the Data Base
System can be viewed as a series of requests and replies, with the system making some requests and the user making others. The first dialog is the identification procedure. The identity and access types of the user are established and retained by the system for future use. (Most contemporary systems employ techniques such as passwords for the identification process. Such "handshake" techniques have obvious drawbacks. Current research suggests that eventually we may supplant these techniques with measurements of some "non-forgeable" characteristic, such as fingerprints or hand geometry. For the nearer term, other techniques such as magnetically striped credit cards may prove feasible.) Once the identity of the user is determined, the purpose of a Data Base System is to use the information in the Logical Data Base in the reply to the user's requests. For every such request the system performs an authentication procedure to determine whether the request should be permitted or denied, utilizing the information established and retained by the identification procedure and other information which describes the protection rules currently in effect. The system then carries out authenticated requests. As more sophisticated Data Base Systems are developed where the separation of Logical Data Base and Physical Data Base is evident, more of the burden of searching for the information in the Logical Data Base
which satisfies the user's request is placed on the system.

To allow users to access only certain parts of a database, the division of the database into logical regions may be required. Furthermore, the Logical Data Base may take on different apparent regions for different users. The study of logical divisions, their access requirements, and their physical organizations is termed compartmentalization, in which parts of the database are separated logically and/or physically. For particularly sensitive parts of the database, it may be necessary to further check on the legitimacy of attempted accesses. These verification procedures may be as simple as asking the user "ARE YOU SURE?", or may be much more elaborate. It is well to compartmentalize even if especially sensitive data is not involved. With these "firewalls" the entire database is less likely to be affected if some untoward event (such as physical damage, accidental destruction, or illegal access) occurs.

It is also reasonable to consider the information used in the identification, authentication, compartmentalization and verification procedures as part of the database which may be manipulated in the same manner as the rest of the database. An authorization procedure is the only means by which this information is created and maintained. In particular, certain users (say, the creator) of logical data
can authorize other users of the Data Base System to access the data by exercising the authorization procedure.

With this discussion, we have the following important notions.

* Security is the prevention of unauthorized use of the data.

* Integrity is the prevention of unauthorized or accidental destruction or modification of the data.

* Access control is the process of determining the authorized users of the Data Base System (and thus the Logical Data Base), and of determining which accesses may be permitted and which should be denied.

* Contamination is a breach in integrity; penetration is a violation of security. Collectively, these are termed interference.

* A protection mechanism (or simply, protection) is an attempt to provide security and integrity by means of access control and interference prevention.

What has been characterized is a general Data Base System with access control and interference prevention. Practical systems embody the procedures discussed above in
various forms reflecting the needs and purposes of the system. Every Data Base System must have some security and access control if the user is to place any confidence in the data stored and retrieved. What distinguishes a Data Secure System is that the security and integrity are integrated into the system, not hung on as an extra module or two. The protection mechanism of the Data Secure System is logically complete and can be convincingly demonstrated to operate correctly and effectively even when under strong attack by well equipped (skillful and knowledgeable) penetrators.

The preceding discussion has laid the formal basis for the conceptual model and has also outlined in a somewhat less formal way the concepts of most contemporary Data Base System protection mechanisms. Let us first go strongly into the theoretical aspects of the conceptual model, and then show how such theory may be applied by demonstrating how some fairly diverse protection mechanisms may be described by the model.

**Defn:** The **Extended Logical Data Base** (ELDB) is a set of triples

\[(u,a,d)\]

where \(u\) is a user identifier,
\(a\) is an access-type identifier,
\(d\) is the identifier of an element of the Logical
The Extended Logical Data Base is formed by making a triple for every possible access to the Logical Data Base by every possible user. Thus, the Extended Logical Data Base is a complete characterization of all possible accesses to the data base. Clearly, the Extended Logical Data Base has an immense number of elements for even the most trivial cases. Our goal is to suggest methods which deal with aggregates of Extended Logical Data Base elements. The Extended Logical Data Base forms the foundation for any discussions of protection.

Let us consider the components of these triples in some detail. Not too much need be said about the user identifier, since its meaning and importance are obvious. However, our definition of access was quite broad so as to include all actions in which information from the Logical Data Base is used. Thus, we must consider the connotations of particular access types. A serviceable definition is, "Each access type is a program type identifier which effects a particular variety of access..." [PopeG73a]. The access type identifiers in the triples are, thus, program identifiers. These access programs range from basic hardware operations through supervisor services to more elaborate user created programs. Finally, we have made the
assumption that every element of the Logical Data Base can be identified and assigned a unique name. These names are the data identifiers of the Extended Logical Data Base.

**Defn:** A protection specification is a function:

\[ p: S \rightarrow \{\text{permit, deny}\} \]

where \( S \) is a subset of the ELDB.

Given \( x \) a member of \( S \), \( p(x) = \text{deny} \) will indicate that under this protection specification, the access attempt would be denied and \( p(x) = \text{permit} \) that the access is permitted.

Intuitively, a protection specification is an assertion about the protection of the Data Base System. We do not require that a user who is creating protection specifications have global knowledge of the system. He can make the specification cover only that Extended Logical Data Base subset about which he is cognizant.

**Defn:** A protection pattern is a set, \( P \), of protection specifications such that every triple of the Extended Logical Data Base is in the domain of at least one specification.

**Defn:** A protection pattern, \( P \), is consistent if \( P \) is itself a function, i.e., every triple of the Extended Logical Data Base which lies in the domain of more than one of the protection specifications of \( P \) is mapped onto the same value by each of the protection specification in
whose domains the subset is contained. A protection pattern which is not consistent will be said to be inconsistent.

We shall assume that protection patterns are consistent unless specifically indicated otherwise. (Later, we shall suggest methods to resolve inconsistent protection patterns.) In a consistent protection pattern it does not matter which protection specification is applied to an element for which several specifications are applicable. By definition if access to the element, is, for example, denied as a result of the application of one of the protection specifications (i.e., $pi(x) = \text{deny}$), the access to the same elements will still be denied no matter which other protection specifications are applied. We may simply refer to $P(x)$, rather than some particular $pi(x)$.

Thm: A consistent protection pattern partitions the Extended Logical Data Base into those accesses which are permitted (may be completed) and those which are denied (indefinitely delayed).

This theorem is the heart of the conceptual model. It formalizes the notion that of all the "things" that the users might try to do, the protection mechanism, at any instant, partitions the "things" into a set which are allowed, and a set which are not allowed. This formalism
I let us consider all deterministic protection mechanisms in the same framework because the result of any protection mechanism is to return a binary (permit or deny) result to a user access attempt. Let us demonstrate how some of the existing protection mechanisms may be characterized by the model.
**Modelling Existing Protection Mechanisms**

Example 1: Modelling Protection Mechanisms Based on an Access Matrix [LampB71, GrahG71, GrahG72].

The access matrix used by the protection mechanism may be conceptualized like the following figure (from [GrahG72]):

![Access Matrix Diagram]

**Figure II.1 - Portion of an access matrix.**

The access attributes (permitted access types, in our terminology) of subject (user, in our terminology) s[i]
towards object \(o[j]\) are contained in the \([i, j]\) entry of the access matrix. For ease of discussion, let us consider a system environment in which we are only concerned with accesses towards files (e.g., \(F_1\) and \(F_2\) in Figure II.1). Further, let there be only four different access types: read, write, update, and delete as shown in Figure II.1.

<table>
<thead>
<tr>
<th>(x)</th>
<th>(p(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s_1, r, f_1))</td>
<td>permit</td>
</tr>
<tr>
<td>((s_1, w, f_1))</td>
<td>permit</td>
</tr>
<tr>
<td>((s_1, u, f_1))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_1, d, f_1))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_2, r, f_1))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_2, w, f_1))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_2, u, f_1))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_2, d, f_1))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_3, r, f_1))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_3, w, f_1))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_3, u, f_1))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_3, d, f_1))</td>
<td>permit</td>
</tr>
<tr>
<td>((s_1, r, f_2))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_1, w, f_2))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_1, u, f_2))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_1, d, f_2))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_2, r, f_2))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_2, w, f_2))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_2, u, f_2))</td>
<td>permit</td>
</tr>
<tr>
<td>((s_2, d, f_2))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_3, r, f_2))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_3, w, f_2))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_3, u, f_2))</td>
<td>deny</td>
</tr>
<tr>
<td>((s_3, d, f_2))</td>
<td>deny</td>
</tr>
</tbody>
</table>

Figure II.2 - A portion of an access matrix expressed as a protection pattern.

Each entry of the table in Figure II.2 is a protection specification. Collectively, the protection specifications form a protection pattern which partitions the data base.
into accessible and inaccessible files. In other words, the table depicted in Figure 11.2 is the protection pattern. This example shows many things. First, the conceptual model can describe the access matrix in a natural, even trivial way. Second, the Extended Logical Data Base has an impossibly large number of elements. Finally, it should be noted that access matrix based protection mechanisms as discussed by Lampson in [Lamp71] and by Graham and Denning in [GrahG71, GrahG72] were mainly concerned with the protection in operating systems where the file-like objects were comparatively few in number and were the same for every user (i.e., protection was limited to the Physical Data Base). In general neither of these two assumptions is true for Data Base Systems, which deal with the Logical Data Base.

Example 2: Modelling Protection Mechanisms Based on Capability Lists.

In essence, a capability list (c.list) is one row from the access matrix. Many different definitions for c.lists exist. We, rather arbitrarily, will use the one suggested by Dennis and Van Horn in [DennJ66]:

Each capability in a c.list locates by means of a pointer some computing object and indicates the actions that the computation may perform with regards to that object. Among these
capabilities there are really several (memory) segment capabilities, which designate segments that may be referenced by the computation and that give by means of access indicators an indication of the kind of reference permitted.

Although the definition of c.list confines the protection to memory protection, we will allow more liberal interpretation of the term "segment". Let us consider these segments as files. The characterization of a c.list-based protection mechanism by the conceptual model is again relatively trivial. Considering only the files, let us first characterize the access matrix in Figure II.1 as a c.list.

![Diagram of access matrix](image)

**Figure II.3** - The rows of an access matrix as characterized by capability lists (c.list).
The c.list system may be modified to compress out the null entries. However, this has no effect on the ideas of either the c.list or the conceptual model. To map c.lists to the protection patterns of the conceptual model, we perform the following procedure. Everywhere that the c.list gives the user a capability to reference some segment, locate the Extended Logical Data Base element (consisting of identifiers of the user, access type and segment) corresponding to that segment. There will be such an element because we have explicitly defined the Extended Logical Data Base to consist of all possible accesses. Next assign the value "permit" to the element (i.e., \( p(x) = \text{permit} \)). In other words, we have just formed a protection specification of the element to which access is permitted. When all the c.lists have been exhausted, we have obtained all the possible elements to which accesses are allowed. Finally, assign all the other non-referenced elements the value "deny" (i.e., \( p(x) = \text{deny} \)). Thus, the non-referenced segments are not accessible. We have now a protection pattern in the conceptual model corresponding to the c.list.
Example 3: Modelling Protection Mechanisms Based on Authority Items.

One of the first systems to introduce logical access control, subfile protection and user-created control procedures is discussed by Hsiao in [HsiaD68a, HsiaD68b]. The basic system protection mechanism uses capability lists which are called authority items. One authority item is associated with each user. The file of authority items is itself maintained much like other files on the system. The system is notable in its ability to offer protection of arbitrary subfiles. These subfiles are defined not by physical parameters but rather by logical descriptions, such as Boolean and arithmetic expressions of keywords and symbolic names. Within the authority item, logical expressions indicate for each file which records are inaccessible, which are temporarily blocked, which are presently opened for use, etc. As a record is retrieved in response to a user query, it is processed against these logical expressions to determine whether it should be output to the user. Figures II.4 and II.5 (from [HsiaD68b]) illustrate these system features. The user is permitted to create a procedure associated with the file which he owns. This procedure would be invoked whenever access to the file is initiated by any user. Such procedures can be
arbitrarily complex, the only constraint being that the procedure return a 1 or 0 to the system indicating whether the access to the file is to be permitted or denied. This procedural idea is expanded upon by Hoffman in [HoffL70, HoffL71] and subsequently became known as formularies.
A File

Records Permanently Protected from Access

Records Belonging to Open Portion of the File

Records that are Temporarily Blocked from Use by Others

Accessible Records

Inaccessible Records

Figure II.4 - The sets of records in a file specified by three types of logical description.
Any More Records to Retrieve in this File?

Any More Opened Files?

Exit

A Record Is Retrieved From an Opened File

Does It Satisfy the User's Query Expression?

Does It Satisfy the Expression for the Protected Portion of this File?

For the Protection of Privacy

For Temporarily Blocking the Access

For Other Access Control Purpose

Record Does Not Satisfy the Additional Condition in the Query

Record Is Classified

Output the Record

Service: A Request for the Retrieval of Records

Figure II.5 - Expression validations in record access.
Example 4: Modelling Protection Mechanisms Based on Formularies.

Perhaps the most difficult protection mechanism to be characterized in the conceptual model are the ones based on the user-defined authentication/verification procedures. These procedures are termed *formularies* by Hoffman in \[\text{HoffL70, HoffL71}\] and are also adopted by the CODASYL Data Base Task Group in \[\text{CODAS73}\]. Essentially, a user could create whatever procedures he would like, and is allowed to have such procedures invoked at various phases of data base activity. Thus, access to data is determined by these procedures. One reason for the difficulty in characterizing procedural mechanisms in the conceptual model is the lack of definition of the procedure (i.e., formulary) itself. It is also not clear what is the environment for the invocation of the procedures. By this we mean what are data that are accessible to the procedure in order for the procedures to determine whether to permit or deny an access attempt.

Fortunately, the conceptual model is only descriptive in nature. Thus, we are not concerned with how a decision to permit or deny access is reached; rather, we only want to know what such a decision is. In effect, we could build the protection pattern equivalent to a given procedure
(formulary) by "running" (or executing) the procedure on every element of the Extended Logical Data Base. Intuitively then, we can characterize procedures (formularies) in terms of the corresponding conceptual model protection patterns.

Example 5: Modelling Protection Systems Based on Rights.

A. Jones [Jones73] has developed a model based upon what she has called rights. Although the most significant impact of her work cannot be discussed until somewhat more of our model has been developed, we introduce her model for completeness.

An environment is a table of rights each expressed as an (object-name, access-name) pair restricted so that an access is named in a right only if it is applicable to the object named in that right... The activity of a process is restricted by the enforcement rule which states that a process can access an object only when the right to do so is in the environment of that process.

Her model is quite similar to ours, the chief difference being that we suggest that the definition of potential accesses be separated from the ability (or lack of ability) to perform them. This separation is achieved by the introduction of the Extended Logical Data Base and the protection pattern. This separation is an important extension as a conceptual tool because it allows us to view
protection macrocosmically as a set of rules rather than microscopically as a set of individual access decisions. As is obvious from the earlier examples, it is practically impossible to consider a protection pattern as a set of permitted and denied accesses. Because we have emphasized that a protection pattern is made up of protection specifications, we can focus some of our attention upon the rules rather than the detail which results from their application.

Four Types of Protection Specifications

Throughout the preceding discussion we have ignored two areas because we lacked the theoretical tools to deal with them. First, how do we resolve the protection of subsets of the Extended Logical Data Base which are assigned different protection by different protection specifications? Second, how do we get an orderly characterization of the creation, alteration and destruction of protection specifications and patterns? In order to treat these questions in a thorough way, we again develop some definitions.

Define: The user extraction operator \( U \) returns for an element \( x \) of the Extended Logical Data Base the user identifier of the element.

\[ U(x) = \text{user identifier} \]
**Defn:** The **access type extraction operator** $A$ returns for an element $x$ of the Extended Logical Data Base the access-type identifier of the element.

$$A(x) = \text{access-type identifier}$$

**Defn:** The **data extraction operator** $D$ returns for an element $x$ of the Extended Logical Data Base the data identifier of the element.

$$D(x) = \text{data identifier}$$

The following notation will be used in this section. Lower case $x$ and $y$ will denote elements of the Extended Logical Data Base. Other lower case letters used as functions denote protection specifications. Thus, $p(x)$ means the protection specification $p$ applied to the element $x$. Capital letters used as functions denote protection patterns. Recall that the difference between a protection specification and a protection pattern is that the specification is a local statement about the protection of a subset of the Extended Logical Data Base while a pattern is a global statement which describes the protection for all possible accesses.

Now let us consider relations between protection patterns.

**Defn:** Two protection patterns, $P$ and $Q$ are **equal**,

$$P = Q \text{ if } \forall x (P(x) = Q(x)).$$
In order to define a partial ordering on the collection of all possible protection patterns for a given Extended Logical Data Base, we will say that for a, b ∈ {deny, permit}, \( a \leq b \) means \( a = \text{deny} \) or \( b = \text{permit} \). Then we say that a pattern \( P \) is a restriction of pattern \( Q \), \( P \leq Q \), if \( \forall x (P(x) \leq Q(x)) \). Informally, \( P \leq Q \) says that

a) There are some of the Extended Logical Data Base subsets which are permitted under \( Q \) but denied under \( P \).

b) There are no subsets which are permitted under \( P \) but denied under \( Q \).

Restriction is a partial ordering on the "strength" of the patterns, that is

a) \( P \leq P \) for any protection pattern.
b) \( P \leq Q \) and \( Q \leq P \) implies that \( P = Q \).
c) \( P \leq Q \), \( Q \leq R \) implies \( P \leq R \).
Given two protection patterns $P$ and $Q$ we define the greatest lower bound (glb) of $P$ and $Q$, $P' = \text{glb}(P,Q)$ as follows:

$$P'(x) = \begin{cases} 
\text{deny, if } P(x) \text{ or } Q(x) = \text{deny.} \\
\text{permit, if } P(x) = Q(x) = \text{permit.} 
\end{cases}$$

Similarly, the least upper bound, $P'' = \text{lub}(P,Q)$, is defined as follows:

$$P''(x) = \begin{cases} 
\text{permit, if } P(x) \text{ or } Q(x) = \text{permit.} \\
\text{deny, if } P(x) = Q(x) = \text{deny.} 
\end{cases}$$

The glb and lub can also be defined for specifications. If $p$ and $q$ are specifications, $p' = \text{glb}(p,q)$ is specification such that:

$$p'(x) = \begin{cases} 
\text{deny if } p(x) \text{ or } q(x) = \text{deny.} \\
\text{permit if } p(x) = q(x) = \text{permit.} \\
\text{undefined otherwise.} 
\end{cases}$$

and $p'' = \text{lub}(p,q)$ is defined as follows:

$$p''(x) = \begin{cases} 
\text{permit if } p(x) \text{ or } q(x) = \text{permit.} \\
\text{deny if } p(x) = q(x) = \text{deny} \\
\text{undefined otherwise.} 
\end{cases}$$

The glb and lub characterize two constructive operations since neither is necessarily equal to either $P$ or $Q$. Thus both glb and lub represent possibilities for resolution of protection in patterns which are not consistent. This resolution is how we can create a consistent (global) protection pattern from potentially inconsistent (local) protection specifications. In attempting to make a consistent pattern, we may be fortunate
enough that the individual specifications are consistent and that they cover the entire Extended Logical Data Base. It is more likely that one or both of these conditions does not hold. First, let us consider the situation in which some of Extended Logical Data Base subsets are in the domain of no protection specification. Here, an essentially arbitrary decision on whether to permit or deny such accesses must be made. Such a decision creates, in effect, a default protection specification covering these subsets. (Saltzer [SaltJ74] points out the "principle of least privilege" as used in MULTICS requires that these accesses not covered by any protection specification should be denied.) The problem of two or more specifications giving the same subset different inconsistent protections is far more difficult. The best working hypothesis is to suspend any access and refer such problems to a higher authority. This restrictive strategy corresponds to taking the glb of the contending specifications because if they differ, it is the case that one says to permit some access, and the other says to deny it. Intuitively, it seems best to suspend the access while waiting for the higher authority to decide, because such actions can be simply reversed. On the other hand, once access has been incorrectly permitted, there is little that can be done to reverse the action. Such a higher authority may well be one of the users. It should be remarked that
the resolution of inconsistent protection specifications is really the more philosophical question of how do we arbitrate between two conflicting requirements in system design. Each of the specifications is an assertion about the desired protection of the system. For this reason, no mechanical procedure for such resolution was given. The method suggested represents a compromise. By suggesting that some decisions can only be made by mechanisms and people outside the model, we endow it with great flexibility. Those decisions about which no controversy exists will be made mechanically and routinely.

All of the above formalism would be rather uninteresting if it could not be related to the practical considerations of system design. We are attempting to create a formal model for description and discussion of a wide variety of protection schemes. It is easy to lose sight of this and fall into a pedantic discussion of esoteric properties of the model. We have developed the notion of the protection pattern as the central descriptive means. By showing protection patterns to be partially ordered under restriction we were able to motivate the glb and lub as operators which could be used for the resolution of inconsistent protection patterns.
Now, we shall consider a more practical question, how do we create and modify the specifications which make up the patterns?

**Defn:** A **context-free protection specification** is a protection specification in which the permit/deny decision depends only upon the \((u,a,d)\) triple describing the access.

Initially, we restrict our discussions to such memoryless specifications, since they are the most basic type. We shall briefly develop some protection specifications which will be used in the sequel.

The most primitive specification, known as **TYPE.1**, gives the protection of a single Extended Logical Data Base element, \((u,a,d)\):

\[ \text{TYPE.1}(u,a,d), [\text{deny/permit}] \]

(the notation \([\text{deny/permit}]\) means choose one of the enclosed terms)

It should be obvious that, formally, this single operator is sufficiently powerful. Other operators will, thus, be measured against this one for flexibility. However, it should also be obvious that this operator is an awkward way to specify the protection of more than a few elements. Moreover, to create such specifications, the user must know explicitly all the users and access types to be governed by
the specifications, an undesirable feature which violates our assertions about global knowledge.

The next step in flexibility, called Type 2 specifications, is to allow the specification to cover a subset, \( S \), of Extended Logical Data Base, rather than a single element.

**Type 2 \((S, \{\text{deny/permit}\})\)**

We observe that in defining a subset there is the practical problem of describing the subset to the protection system. Certainly, we shall not want to describe the set by enumerating its elements, since we are no better off than with the Type 1 protection specification. We can ameliorate this situation by restricting the type of subset to one which can be described by parameterization. For example, the set definition \( \{x : x \geq k\} \) implicitly creates a set selection function with parameter \( k \) indicating whether or not the element is a member of the set. Although we would be somewhat premature in specifying at this point the set selection functions which we shall use, we nevertheless stipulate that whenever a subset \( S \) is used in a protection specification, there is a set selection function of a few parameters associated with \( S \). We shall continue this assumption throughout the rest of our discussion on the conceptual model.
The TYPE.2 specification enables the user to specify that every element of S has the same protection. What we would now desire is to relax this by allowing the user to specify that some subset S is to be protected "like" some other subset S'. This type of specification will be called TYPE.3. First we must define what we mean by "like". Let f be a function from S into S', two Extended Logical Data Base subsets. Then for x in S we make p(x), the new protection specification for x, to be P(f(x)), the currently defined protection pattern as applied to f(x).

Since f maps S into S', the specification can actually be simplified to

TYPE.3 (S, f).

An example of one such function is:

\[ f: d' \quad ((u,a,d)) = (u,a,d') \]

With such a function we can permit and deny the same set of accesses to two different data elements. That is, if P((u,a,d')) = permit, then we make p((u,a,d)) = permit. For the same user u the data element d is protected with the same access attributes as the data element d'. Obviously, we would like to extend this idea to other elements of the triple, for example, to give one user access privileges identical to those of another user, or to say that users may have access to some data element only if they have a
access to the element. The following general function can be used for all of these situations.

\[
f(x; u', a', d') = \begin{cases} 
U(x) & \text{if } u' \text{ is null} \\
u' & \text{otherwise}
\end{cases} \begin{cases} 
A(x) & \text{if } a' \text{ is null} \\
a' & \text{otherwise}
\end{cases} \begin{cases} 
D(x) & \text{if } d' \text{ is null} \\
d' & \text{otherwise}
\end{cases}
\]

where

\[x \in \text{Extended Logical Data Base, i.e. a (u, a, d) triple.}\]

\[u' = \begin{cases} 
\text{null} & \text{user identifier} \\
\text{null} & \text{access type identifier}
\end{cases}\]

\[a' = \begin{cases} 
\text{null} & \text{data identifier}
\end{cases}\]

Let us consider the following special cases of \(f\) for \(x = (u\emptyset, a\emptyset, d\emptyset)\), an element in Extended Logical Data Base.

Case 1:

\(f (x; \text{null, null, } d\emptyset) = (U(x), A(x), d\emptyset).
\)

But \(U(x) = u\emptyset\)

\(A(x) = a\emptyset\)

so that

\(f (x; \text{null, null, } d\emptyset) = (u\emptyset, a\emptyset, d\emptyset)\)

Intuitively, this function indicates that for the user \(u\emptyset\) the data element \(d\emptyset\) can be accessed in the same \(a\emptyset\) manner as data element \(d\emptyset\).
Case 2:

\[ f(x; u_2, \text{null, null}) = (u_2, A(x), D(x)) = (u_2, a_0, d_0) \]

This function says that user u0 will have the same a0 access rights to data d0 as user u2 has.

Case 3:

\[ f(x; u_1, a_1, \text{null}) = (u_1, a_1, d_0) \]

This function says to make user u0's a0 access to data d0 the same as user u1's a1 access to d0.

It is, of course, possible to define many such functions. But first let us show how some common protection requirements can be translated into protection specifications of TYPE.3. Typical systems have a default protection for newly created objects [DECSYS1]. One of the strengths of our model is that a wide variety of methods can be used to achieve the same end. One way to get the default protection would be to specify explicitly whether each access was to be permitted or denied, using TYPE.1 specifications. Another way would be to group the permitted and denied accesses into subsets and use TYPE.2 specifications. This subset grouping would be fairly easy since, in general, the default rules are quite simple. The most natural way is to introduce another artifice. For each user we shall consider an artificial, default data object,
d.default, such that the protection of newly created objects is "like" that of the default object, unless otherwise specified. More formally we have the following specification:

\[
\text{TYPE.3} \quad \left( \text{xid}(x) = \text{d.new}, f(x; \text{null}, \text{null}, \text{d.default}) \right)
\]

That is, give every access of the form \((u, a, \text{d.new})\) the same protection as the corresponding access \((u, a, \text{d.default})\). The default specificataion can be changed, should the user desire, affecting only subsequently created objects. We can now close a potential loophole. The tacit assumption was made that the function mapped elements to other elements whose protection was defined. If we use this default idea for each newly created object, there will be no accesses for which the protection is undefined.

Let us consider one more type of specification which will be called \text{TYPE.4}. If we allow multiple valued mappings some additional possibilities occur. A user can create specifications that say "permit this access if any of these other accesses is permitted". We shall use the following new specificatation

\[
\text{TYPE.4} \quad (S, F, \text{OP})
\]

where \(S\) is an Extended Logical Data Base subset, \(F\) is a set of functions \((f_1, f_2, \ldots)\) mapping the Extended Logical Data Base into itself, i.e.,
43

f1: ELD B -> ELD B, and

OP is glb or lub. The specification operates as follows:

\[ p(x) = \text{OP}(p(f1(x)), p(f2(x)), \ldots) \]

recall that the glb will deny an access if it was denied under any of the specifications, and lub will permit it if any of the specifications permitted it.

It is certainly possible to define other, more elaborate, specifications. However, one reaches a point of diminishing returns in the ability to apply them. One also begins to get into the work being done by Hartson [HartH75] on languages for specifying protection.

Summary

In summary, this chapter presents a formal model for protection systems. The model is largely descriptive in character. The operators introduced are to allow us to consider how differing design specifications may be resolved. A variety of other protection models can be incorporated into the framework developed here. A significant point of the model is that it is based upon the logical data resources of the system, rather than physical considerations. In succeeding chapters we will refine this.
conceptual model, eventually ending with the description of a demonstration system which has actually been implemented.
Chapter III. The Structural Model

Introduction

Having laid a definitional foundation in the conceptual model, let us now consider the bridge between it and the realities of implementation. This intermediate model will be termed the structural model because its major concerns are modelling the structure of the data base and the associated accessing programs. The goals of the structural model are to suggest a framework in which demonstrations of completeness and correctness are easily accomplished. To show the model is complete, we shall demonstrate how any protection pattern can be naturally achieved. For correctness, we shall establish that necessary and sufficient conditions are met which insure that any access denied by the protection pattern will be indefinitely delayed.

In describing the structure of a system, we are actually concerned with two distinct but strongly related structures, the data structure (passive element) and the
program structure (active element). In the area of programming languages, for example, the choice of which language to use for a given task is most strongly influenced by its generic data structure capability. The earlier COBOL vs. FORTRAN debates stand in support of this contention. Recently, the active component of systems has come into sharper focus. The furor over "structured programming" is evidence of the concern which the community is now investing in the dynamic component of system design. In the present work we shall consider the data base structure in the data base model and the system structure in the system model.

The Data Base Model

The preceding development of protection specifications made no explicit use of the fact that we are dealing with data base systems. It is here that the present work makes a sharp departure from other efforts which were primarily concerned with protection in operating systems and is thus a new contribution to the state of the art. Though certainly no less significant, the problems of protection in operating systems are different from those of data base systems. First, even a large, multi-user operating system is concerned with the protection of a small number (like hundreds or at most a few thousand) of relatively large
objects, the majority of which are some kind of physical resource such as disk files and memory segments. A database system is concerned with large numbers (tens of thousands to millions) of relatively small (tens to hundreds of words) objects such as the records and fields. Second, in an operating system the objects are mostly unrelated to each other, and where such relations exist between objects, they are fairly simple. In modern database systems a large variety of relations exist between objects. Indeed, this is a fundamental purpose of database systems, to allow the retrieval of information from many different objects based upon some relationship or affinity among those objects. Thus, a database protection system should, even must, use those relations to provide or enhance security.

In order to emphasize the relationships among objects of database systems, a more formal model is needed. Although the topic of database modelling has been an extremely popular one in recent years, reflecting the very real need for such formalism, we do not need the complexity which characterizes many of the models proposed. The following terminology and ideas are partially derived from [HsiaD70, WongE71, McCaE75].

We start with two undefined terms: a set \( A \) of "attributes" and a set \( V \) of "values". These are left
undefined to allow the broadest possible interpretation.

**Defn:** A record $r$ is a subset of the Cartesian product $A \times V$, in which each attribute has one value. We can consider $r$ to be a collection of ordered pairs: (an attribute, its value). Every record is assigned a unique address.

For practical reasons we shall limit our consideration to (attribute, value) pairs which are succinct. Such pairs will be called keywords, and will be denoted by $K$ or $K_i$. Although larger pairs will still be allowed to occur in records and will be returned when the record is retrieved, only keywords may be used in queries.

**Relations on records of the data base**

**Defn:** A relation on the records of a data base is a set of ordered pairs of records $(r, r')$. If two records are related by a relation, REL, we denote this be $r \REL r'$.

**Defn:** REL is an equivalence relation if it has the following properties:

1) reflexive: for any record $r$, $r \REL r$
2) symmetric: if $r \REL r'$ then $r' \REL r$
3) transitive: if $r \REL r'$ and $r' \REL r''$ then $r \REL r''$
An equivalence relation partitions the records of the database into disjoint groups called equivalence classes. For a given equivalence relation $\text{REL}^*$, only records within a single equivalence class are related to one another by $\text{REL}^*$. Two records in different equivalence classes of $\text{REL}$ are not related by $\text{REL}$. (They may be related in some other way, however.)

We shall now identify a number of useful equivalence relations on the records of a database. The first relation is derived from physical considerations.

**Cell Equivalence**

We will idealize the notion of the logical address space of the database. In this idealized version, the logical address space is viewed as being non-homogeneous, being divided into secondary storage cells (or more simply cells). Cell boundaries occur every $s$ addresses. We shall refer to the integer $s$ as the **call size**. Records within one cell are "close" to each other and are "far away" from records in other cells. More formally we make the following definitions.

**Defn.** Two addresses $a$ and $a'$ are related by $\text{CEA}$, cell equivalence if
\[ a = e \times s + d \]
and
\[ a' = e \times s + d' \]
where \(0 \leq d < s\) and \(0 \leq d' < s\)

The number of cells is one greater than the maximum value of \(e\). We have continued the practice of using the neutral term cell. Depending upon the needs of the system designer and the limitations imposed upon him, a cell may correspond to a track or a cylinder on a disk, a page in a virtual memory system, etc. In general, the first access to a given cell is more "costly" than subsequent accesses to the same cell. The cell equivalence relation on addresses, CEA, determines an equivalence relation on records, CER, based upon the physical placement of the records. Two records are related by CER if their addresses are related by CEA, i.e., the two addresses are in the same cell. (It should be noted that some operating systems tend to hide the physical cells from the user, presenting him instead with a homogeneous logical address space. In such systems, direct realization of the cell equivalence relation may not be possible.)

**Logical Equivalence**

Another category of equivalence relations is based upon the logical content of the record as revealed by its
keywords. In earlier work, both Wong and Chiang [WongE71] and Rothnie and Lozano [RothJ74] have discussed such logical equivalence relations. We are significantly generalizing both pieces of work and are unifying their two diverse concepts.

**Defn:** A keyword, \( K \), is **true** for a record, \( r \), if \( K \) is in \( r \), otherwise it is **false**. A **query** is a proposition given by a Boolean expression of keywords. A query is **true** for a record, \( r \), if this proposition holds for the keywords of \( r \). In this case, \( r \) is said to **satisfy** the query. Thus, every Boolean expression \( f(K_1, \ldots, K_m) \) is either true or false for each record.

**Defn:** Let \( X=\{K_1, K_2, \ldots, K_n\} \) be a set of keywords. We shall refer to this as the **basis** of the logical equivalence relation.

The logical equivalence classes are defined by the \( 2^n \) Boolean expressions of the form:

\[
K_1^* \land K_2^* \land \ldots \land K_n^*
\]

where \( K_i^* \) is \( K_i \) or \( \overline{K_i} \)

Clearly, each record will satisfy one and only one of these expressions. The collection of these functions defines an equivalence relation on the records. Each expression is a set selection function for a potential logical equivalence
class. In the sequel we will tend not to make the subtle distinction between the canonical (minterm) expression and the set of records belonging to the corresponding equivalence class. Where the distinction is important to the point being made, it will however be emphasized.

Wong and Chiang [WongE71] have considered at length the situation in which all the keywords in the file are in the basis. What they have shown is that in such a case, the non-empty logical equivalence classes are atoms of the collection of all retrievable sets of records. This collection is a Boolean algebra, and will be referred to as $B(Q)$.

**Defn:** $T$ is an atom of a Boolean Algebra $B(Q)$ if

1) $T \in B(Q)$

2) $T$ is non-empty

3) No non-empty proper subset of $T$ is in $B(Q)$.

**Thm:** Let $T_1, T_2, \ldots, T_j$ be the atoms $B(Q)$, then

1) $T_i \in B(Q)$ for $i = 1, 2, \ldots, j$

2) $T_i$ and $T_k$ are disjoint if $i \neq k$

3) $U \in B(Q)$ implies that for each $i$, $U \cap T_i$ is either empty or is $T_i$.

4) Every $U \in B(Q)$ is the union of some of the $T_i$.

We stated earlier that it was possible to enumerate every answer to any question that the user could put to the
data base. This result can be achieved by characterizing any arbitrary subset of the Logical Data Base as a collection of atoms. Furthermore, it is possible to completely characterize a protection pattern as either the subset of permitted or of denied accesses. Thus it is possible to specify protection patterns on any subset of the Extended Logical Data Base by describing which atoms are permitted and which atoms are denied for each access type. Every retrievable item in the Logical Data Base may be protected since retrieval specifications are like TYPE.3 specifications. Although this statement is deceptively simple, its implications are great. We are now talking about the ability to characterize explicitly the protection of logical entities rather than their physical realizations. The most significant impact of such formalism is that it allows us to verify the assumption that we can find every piece of data in the Logical Data Base, rather than restricting our protective abilities to the Physical Data Base.

**Structural equivalence**

A third type of equivalence relation is structural. The structural equivalence relation, STR, models those partitioning decisions which are based upon factors other
than the record's address or its keywords.

**Defn.:** Two records \( r \) and \( r' \) are related by STR, structural equivalence, (i.e. \( r \) STR \( r' \)) if it is possible to determine the location of \( r' \) using only the file linkage information in \( r \) and other intermediate records along the path to \( r' \), or if record \( r \) can be located from \( r' \) in the same way.

(Because the discussion of the file structure must be delayed until an additional equivalence relation is developed, we cannot give a precise definition of what is meant by "file linkage information" until slightly later.)

The most important of these other types of partitioning result from design decisions about the file structure. For example, in an inverted file the designer, in effect, specifies that no record is related to any other record by the file structure, i.e., the equivalence classes of STR have exactly one member each. It is also desirable to limit the size of the STR equivalence classes for other reasons, such as to minimize the propagation of damage. If, for example, all the records of the file were on a single chain, i.e., there is one STR equivalence class with all the records in it, damage to any record might render unreachable all those records further down the chain. In some ways STR is similar to CER. We emphasize the separation because STR
models the results of design decisions, and CER models the results of physical device characteristics.

**Cluster equivalence**

The three equivalence relations already discussed, CER, LOG, and STR may be intersected to form a new equivalence relation. This equivalence relation, which we shall call cluster equivalence, CLS, has the property that two records \( r \) and \( r' \) are related by CLS (i.e., \( r \, CLS \, r' \)) if and only if \( r \, CER \, r' \), \( r \, LOG \, r' \), and \( r \, STR \, r' \). We shall call the equivalence classes of CLS clusters.

Because each of these three equivalence relations (CER, LOG, and STR) is defined for every record, and because they are mutually independent, the resulting cluster equivalence relation, CLS, is non-null and is defined for every record. In the very worst case, the CLS equivalence classes contain one member record each, i.e. an inverted file. In other cases the CLS relation will induce a partitioning that is as "fine" as the "finest" of the partitions induced by CER, LOG, and STR.

The use of the new concept of cluster equivalence allows us to simultaneously reflect desirable logical, physical, and structural properties. We will see that an
insertion algorithm which preserves these logical, physical, and structural properties allows us to base an access algorithm on the cluster equivalence relation with assurance that efficient access will be performed.

When a system user presents a record to be inserted in the data base, the keywords of the record determine the logical equivalence class to which the record belongs. The system decides where the record is to be placed, which determines to which cell equivalence class the record will belong. The record's membership in a structural equivalence class is determined by how the system links it into the data base structure.

File structure

To complete our discussion, we must add to our basic model the data base structure which relates records to each other.

Definition: Associated with each keyword K in record r is the address of another record with the same keyword. We shall call this the pointer of r with respect to K or briefly the K-pointer. We allow the existence of null pointers to retain the uniformity of definitions.
**Defn:** A list $L$ of records with respect to keyword $K$ (or briefly a $K$-list) is a set of records each containing $K$ such that:

1) The $K$-pointers are all distinct.
2) Each non-null $K$-pointer gives the address of a record within $L$ and $L$ only.
3) There is a unique record in $L$ not pointed to be any other record containing $K$, called the **beginning** of the list.
4) There is a unique record in $L$ with a null $K$-pointer, the **end** of the list.
5) No two $K_i$-lists have a record in common for the same $K_i$.

We let $t[i,j]$ be the number of records in the $j$-th $K_i$-list, which has beginning address $a[i,j]$.

**Defn:** A set $F$ of records is called a **file** if every $K$-list containing one or more of these records is contained in $F$. Every file is assigned a unique **file name**.

**Defn:** A **directory** of $F$ is a set of sequences, one for each keyword $K_i$, called **directory entries** for $K_i$, and denoted $E(K_i)$, each of which is of the form:
E(Ki) = \{(Ki,h[i],(c[i,1],a[i,1],t[i,1])),\ldots,
(c[i,h[i]],a[i,h[i]],t[i,h[i]])\}

h[i] is the number of clusters which include
records containing Ki

c[i,j] is a cluster identifier, and
a[i,j] and t[i,j] are as defined above.

We shall require that every K-list have all its member
records in the same cluster. This will be called the
disjoint list property. This property, and the fact that
there is an entry in the directory for each distinct K-list
identifying the cluster in which that list is contained,
allows clusters to be excluded from possible search, if the
cluster does not contain records which satisfy a query.
This exclusion can be done by processing only the directory.
The disjoint list property also allows us to rely upon the
compartmentalization induced by the clusters.

Now we can define more formally what was meant by "file
linkage information" in the definition of the structural
equivalence relation. The linkage information is the
K-pointers. Two records are in the same STR class if they
are on the same Ki-list for some keyword Ki. Because all
the records in a cluster must belong to the same LOG class,
they will have at least one keyword in common. Thus, there
will be at least one Ki-list which threads through all the
records in the cluster, i.e. the records are also in the same STR class. The disjoint list property means that no other records outside the cluster are in the same STR class.

**Defn:** A *generalized file structure* is a file with its directory.

Many common file structures can be viewed as special cases of the generalized structure. That is by suitable choices of s, the cell size; the STR class size limit (i.e. degree of inversion); and the keywords in X, the basis for LOG, one can achieve such diverse file structures as inverted files, indexed sequential files and cellular multilist files, as well as many other types which have not been given generic names.

We introduce a new type of protection specification, **TYPE.5.**

**TYPE.5** (U, Q, [permit/deny])

where U is a set of users

and Q is a Boolean expression of keywords qualified by access type.

Note that this is a logical protection specification, and that the specification is posed completely in terms of the users view of the data base, i.e., Boolean expression of keywords. **TYPE.5** specifications are really quite similar to **TYPE.2** in that we are specifying a subset of the Extended
Logical Data Base by describing the users and a collection of atoms. The separation is emphasized because the TYPE.5 specification is completely in terms of the user's view of the data base. It may be that the set of records satisfying Q is null, that is, at this moment none of the records in the data base are affected by the specification. The decision on what to do in this case may be left to the system designer. He may wish to retain the specification for use in the event that records which meet the specification are subsequently input, or he may wish to ignore the specification, perhaps notifying the user that no records exist meeting the criterion of the specification.

Access Types

As we indicated earlier, to be a useful model, we should be able to accommodate a variety of different access types. The following consideration will lead to a more unified treatment of access types. For each record we allow the possibility of a special (attribute, value) pair, (procedure, Proc) where proc is a procedure. When a record with such an (attribute, value) pair is accessed, control is passed to the procedure. The procedure uses only the information in the other (attribute, value) pairs in the
record. Such pairs always have null pointers.

Thus, the particular access type for a given node is dependent upon the path followed through the data base in getting to that node. This solves the problem posed by multiple access types to the same data element. The particular path followed is, of course, determined by the Boolean expression of keywords supplied by the user. In this way the Boolean expression not only determines the set of records to be accessed but also the types of access to be involved. The following example demonstrates the utility of this formulation.

**Example: Audited Access.**

Consider a situation in which the users can be divided into two disjoint groups, A and B. Those in group A are skilled and trustworthy and are, thus, to be allowed free access to the data base. Those in group B are more suspect. We therefore desire to make a record of every one of their accesses. The data base may be structured as shown in Figure III.1.
The rest of the Data Base

Group B entry → Audit Proc → Directory

Group A entry

Figure III.1 - Audited accesses.

Each user in group B must make all his accesses through the auditing procedure, which has a pointer to the normal search procedure, the directory node. On the other hand, group A may access the directory node directly.

In this new formulation, the directory is not merely an entry into the data base, but also a node at which the normal search procedure is triggered. We term this the directory node. As it is shown in the example, it may be desirable to incorporate more than one entry into the data base. We shall term all these entries as gate nodes. Recalling an earlier discussion of basic terminology, we see now that the function of the identification procedure is to direct the user to the proper gate node. (This notion of access types is quite similar to the formulary approach
discussed earlier, and thus has all its flexibility.)

Another, more subtle, problem which may occur can best be introduced by another example.

**Example:** The pass through problem.

Consider a system with two users, U1 and U2, and the following two records:

```
K1 K2
K2 K3
```

Figure III.2 - Pass through problem.

Let the following protection specifications exist:

- TYPE.5 ((U1), K1, deny),
- TYPE.5 ((U2), K3; deny).

K2 is a legitimate query for either U1 or U2. Yet there is no way to structure a single K2-list so that neither user needs the pointers from a record to which he is denied access, i.e., that no user has to pass through a record to which he is denied access in order to get to another record to which he is permitted access.

Popek [PopeG73a] has considered this problem slightly
in examining the connectedness of the linkage graph if
denied records and their incident edges are removed. We are
being more general in considering whether all legitimate
paths are preserved.

If there is only a single user, this problem does not
occur, since there is always a reorganization of the Ki-list
such that no pass through need occur. This can be done, for
example, by linking the records permitted to the user at the
beginning of the lists and those denied at the end.

In the example, the multi-list pass through problem can
be satisfied by creating two K2-lists, each with only one
record. However, this is not satisfactory for two reasons.
First, it would , in general, be necessary to repeat the
restructuring every time the protection pattern is changed.
In all but the most stable systems, this probably would
result in excessive overhead. Second, the solution to a
similar problem has been shown to be quite difficult.

Ghosh and Eswaran [GhosS72, EswaK73] have considered
the following problem:

Let F be a file stored on a one-dimensional medium
(e.g., tape). The set Q is a pre-specified set of queries
q[i]. Suppose that there is a 1-1 function which maps the
records belonging to F into storage locations of the medium
satisfying: 1) for each query \( q[i] \) in \( Q \) there is a sequence, \( s[i] \), of consecutive storage locations containing all records which satisfy \( q[i] \) and 2) \( s[i] \) does not contain any records which do not satisfy \( q[i] \). Then, \( Q \) is said to have the **consecutive retrieval property** (CRP). Essentially, when we have the consecutive retrieval property once the first record which satisfied the query is located, the subsequent retrievals can proceed very quickly. If the queries in \( Q \) are equally likely to be used, a file with the consecutive retrieval property guarantees minimum retrieval time and storage space. Eswaran [EswaK73] states, "The solution (finding an organization of \( F \) with CRP) for the general case was not found to be 'easy' — easy in the complexity sense." In some ways the problem is analogous to the set covering problem.

The problem of organizing a file to avoid the pass through problem for each user is quite similar to the problem of organizing the file to have the consecutive retrieval property. What we seek is a file organization in which the records that are permitted to each user are "together" in the sense that no denied records are interspersed. Our requirements are slightly different because we need not have physical proximity, rather, we seek an organization with the property that each K-list exhibit
the consecutive retrieval property with respect to a set of queries that are the TYPE.5 protection specifications describing the permitted records for each user, one such query for each user. In the event that for some list no consecutive retrieval organization exists, that list must be subdivided until the resulting lists have the required consecutive retrieval property. (This discussion assumes that users can be directed to a record other than the nominal beginning of a list, a slight extension to Hsiao's original model).

The facts that finding consecutive retrieval organizations is difficult in the first place, and that the process must be repeated whenever the protection pattern is changed weigh against this approach. As an alternative we need a file organization that is immune to changes in the protection pattern and that is relatively easy to generate. One such organization is based upon the LOG (logical equivalence) relation and the concept of atoms previously discussed.

The examination of the work of Wong and Chiang on atoms was motivated by how it could be applied in the verification of the basic assumption that the logical content of a data base could be determined. Now, we suggest that such an organization has benefits in solving the pass through
What gives such an organization the ability to preclude the pass through problem is that an atom cannot be subdivided. Indeed, the definition and the theorem about atoms state "there is no, non-empty proper subset of the atom in the Boolean algebra". The other constructs, files, cells and lists, do not possess this property. Each user is permitted access to all the records in an atom or to none of them. Every list has the consecutive retrieval property with respect to the TYPE.5 queries that describe the protection pattern. When we discussed how the pass through problem was related to the consecutive retrieval property, we suggested that a K-list which could not be rearranged to have the consecutive retrieval property would have to be subdivided. In the most general case, this subdividing would not stop until the atoms had been reached. The atoms do not change with changes in the protection pattern, since they in no way depend upon the particular protection pattern currently in use.

The suggestion that a data base should be organized into atoms would be indefensible if the creation and maintenance of the data base were made extremely complex as a result. Let us briefly examine the magnitude of effort involved in both creation and maintenance. (In the
engineering model, we shall consider these algorithms in more detail and we shall consider the more general case where the logical equivalence partitioning does not include all the keywords of the file.)

Consider a set of records. If there are m distinct keywords then there are $2^m$ possible atoms, one for each canonical expression of keywords. (Later we shall show that most of the canonical expressions cannot be satisfied by any records.) So, the problem of the initial creation of such a file becomes one of finding the non-empty potential atoms. Consider an array with $2^m$ buckets. We could process each record in turn, appending the address of the record to the appropriate bucket in the following way. Define an arbitrary ordering of the m keywords. For each record form an m bit binary number by letting bit i be 1 if $K_i$ is true for the record and 0 otherwise. Use this binary number as an index into the bucket array. When all the records have been processed, the non-empty buckets are the atoms. Naturally, such a scheme could not be actually used. However, it suggests a conceptual method of finding the atoms which requires an amount of effort proportional to the number of records.

The problems of maintenance are somewhat more difficult. Several different maintenance actions should be
considered. Addition of a record with no change in the set of keywords represented in the file is relatively straight-forward. The new record is either in an existing atom or it comprises a new atom. In either case the processing requirements are relatively small. Deletion of a record is also simple. All that needs to be done is to adjust the directory to reflect the deletion of the record, and indicate that the space occupied by the record may be subsequently reclaimed for reuse.

Deletion and addition of keywords also pose fairly simple tasks. A keyword would be deleted if all the records containing it were deleted. A keyword would be added if an added record contained a new keyword. Since we may not wish to control keyword addition or deletion, we probably will not actually be able to use the simple, one bit per keyword, binary representation suggested earlier. This simple representation for each record made explicit use of keywords not contained in that record. This violates the requirements against global knowledge, since to process a single record we would need knowledge of all the extant keywords. One should recognize that neither deleting nor adding keywords can change the existing atoms. Adding a keyword results in a new atom, and forces the existing atoms to include the new keyword (negated) in the conjuncts that
describe these existing atoms. They are otherwise unaffected. Deletion of a keyword would occur if all the records containing the keyword are deleted. This would also mean that there would no longer be any member records in the atoms to which these records belonged, because all the records in the atom contain the deleted keyword. If the keyword has been deleted, all records containing it must have been deleted. No other records are affected.

We should also briefly consider the effects of changing the set $X$, the basis of the logical partitioning. If a keyword is added to $X$, we would have to break-up those clusters which included both records containing the added keyword, and records which did not contain it. One could determine whether or not a cluster had to be broken up by following the $K$-lists associated with the added keyword. Clusters which did not include any records containing the added keyword would be unaffected. If a keyword currently in $X$ is removed from $X$, we may not have to do anything. The existing clusters still continue to satisfy the canonical expressions based upon the revised $X$ (i.e. $X$ with the keyword removed). It may be desirable to coalesce those clusters whose new canonical expressions are now identical, because the original canonical expressions differed only in that the removed keyword was true in one expression and
false in the other. This action is not essential, and in fact, may be precluded by the other equivalence relations, for example, if the STR class size limit is one record.

Thus, none of the maintenance actions is terribly difficult. The discussion of the database model will be now left in favor of the other component of the structural model, the program structure. In the next chapter we shall consider the algorithms to access and maintain the database. To put these algorithms in the proper perspective, we must first consider a higher level model of the program structure.

The System Model

The purpose of the system model is to outline the structure of the active component of a Data Secure System. We shall consider a design in which any protection pattern defined on the previously discussed database model can be demonstrated to be completely and correctly realized by the system. The imposition of the structural model between the conceptual model and the implementation model allows us to base arguments concerning completeness and correctness on overall system architecture. The implementation model then need only show that it actually creates the architecture.
discussed in the structural model.

Any system design is the result of tradeoffs between opposing requirements. In a Data Secure System, the two major contending forces are efficiency and correctness. To provide a meaningful Data Secure capability the system must determine for every user action whether that action violates the protection rules in effect at that time. Since most of this checking cannot be considered to be in response to user requests, it must be viewed as system overhead. The designer of a Data Secure System is faced with the dilemma of maximizing the productive capacity of the system by minimizing overhead yet retaining the security and protection features required of a true Data Secure System.

By the very nature of Data Secure Systems, they possess characteristics which exacerbate this dilemma. The system must be ever vigilant, since any user action could potentially be an attempt to violate the protection rules. The system must be highly reliable, because it must be assumed that potential penetrators are highly knowledgeable and skillful, and are able to take full advantage of the aberrations of unreliable systems. The system must be logically complete. There can be no "trap-doors" etc. All of these aspects seem to imply greater overhead in the form of redundancy and increased analysis of user actions. It
must also be recognized that the majority of user actions are legitimate, and thus, should have only minimal interference.

In the following we shall suggest a model which may be implemented in various ways. Many of the more traditional access control models are merely different implementations of the model to be developed here. This model is based upon multi-accessing deadlocks.

**Defn:** A **deadlock** is a situation in which the progress of two or more processes is mutually and simultaneously inhibited.

The phenomenon of deadlock has been considered by many authors [CoffE71, DijkE68, HabeA69, HaveJ68, HebaP70, HoltR71, HoltR72, KingP73, ShosA69a, ShosA69b]. Historically, deadlocks have been considered mainly as a problem to be avoided by designers of operating systems. These designers must consider two classes of deadlocks. The first class is deadlocks which result from unrestrained contention for system resources: I/O devices, data sets, main memory, etc. The second class results from explicit process interaction, usually from the incorrect use of synchronization primitives. Holt [HoltR71, HoltR72] has presented a unified treatment of both these two classes.
Certain logical conditions must hold for a deadlock to occur. They are:

1. Mutual exclusion. No more than one process may be assigned the use of any resource.

2. No pre-emption. Once a process has been assigned a resource, the resource cannot be reassigned to another process until the original process explicitly releases it.

3. Wait-for. A process holds its previously assigned resources until all of its resource assignment requests have been satisfied. After the requests have been filled, the process will perform its work and release all the resources within some finite time.

4. Circular wait. Two or more processes in a set are each waiting for some other process in the set to complete and release its resources.

Let us digress a moment and illustrate these conditions with a somewhat prosaic example from Coffman and Shoshani [CoffE71].
The situation in Figure III.3 is that we have a town square ringed by one-way streets. The small rectangles represent cars, with the arrow indicating the desired direction of progress. If we consider the cars as being equivalent to user processes, and the "car-sized" pieces of pavement as the resources, this situation meets the four conditions for deadlock.

(1) Mutual exclusion. Clearly, two cars cannot occupy the same piece of pavement at the same time.

(2) No pre-emption. A car cannot force its way into the space occupied by another car.

(3) Wait-for. A car occupies the space in which it is until the space in front of it opens.

(4) Circular wait. Starting anywhere, the car cannot move until the car ahead moves, and so on.
around the square. Eventually we arrive back where we started.

Like any deadlock, this condition will continue to exist until it is broken up by some external force. This traffic deadlock could be broken by the removal of any car and proper sequencing of the flow at each of the corners.

Let us continue with this example to suggest how deadlocks may be used for beneficial results. Imagine that there is a "NO RIGHT TURN" sign on 1st Street at B Street. As the police, we want to enforce this law strictly, but like most police forces we have limited resources and cannot dedicate an officer full time to monitor the traffic at the corner. We desire to have reasonable certainty of preventing the violation of the "NO RIGHT TURN" sign. One way for us to do this is to create a situation in which an illegal right turn will cause a deadlock like that of Figure III.3, yet legal action on 1st Street is unimpeded. We do this by backing up traffic on A, B and 2nd Streets as shown in Figure III.4. An illegal right turn creates a situation like that in Figure III.5 which is identical to Figure III.3
Note that every illegal turn attempt is caught and held in the deadlock. We, the police, can at our leisure come by and arrest the offender, since he cannot proceed away and escape. A significant problem with such a technique for rule enforcement is that to enforce the no right turn rule we inconvenience drivers who are attempting legitimate travel on A, B and 2nd Streets, because these streets must be blocked to achieve a true deadlock condition. Also, when
a violation occurs, traffic flow on all four streets stops.

This example illustrates most of the fundamental features of deadlock-based protection mechanisms.

**Defn:** A **deadlock-based protection mechanism** is a protection mechanism which uses deadlocks to enforce the protection pattern. When a user attempts an access which is denied under the protection pattern, he will go into a deadlock.

By having a violator go into a deadlock, we satisfy the earlier definition of denying an access, "An access is denied if it is indefinitely delayed (never completed)...".

Let us elaborate on what the characteristics of a deadlock-based protection mechanism should be. First, there must be other processes in the system, because a deadlock must involve more than one process. Secondly, we must be in a situation that is almost a deadlock, which we shall call an **incipient deadlock**; one such that the illegal action immediately creates a deadlock where there was not one before. These two conditions will be made more concrete in the subsequent presentation, but now we should get somewhat closer to the data base deadlock problem.

Only a limited amount of material has appeared in the literature regarding the problem of deadlock in data base
systems ([KingP73, BachC73] for example). Although the logical conditions discussed earlier must still hold, they are subject to some other constraints and interpretations.

(1) Mutual exclusion. Of the possible accesses to a Logical Data Base element, only certain accesses are by nature exclusive. Thus, it is reasonable (and in large systems, essential) that data be shared between users whose accesses to it are not in conflict, for example two users who wish to read the same record.

(2) No pre-emption. Rather than being an inherent feature, the pre-emption question is a design decision. Again, it may be reasonable in a particular system to say that certain access types may not be pre-empted, for example updating, and the certain other access types may be pre-empted, for example reading.

(3) Wait-for. We stipulate that a user may make a special type of access to any element of the Logical Data Base. The characteristics of this special access are that it is exclusive and that it is only completed when the user indicates it is. This type of access has been called BLOCK-UNBLOCK by some authors [HsiaD68a, HsiaD68b]. Other authors [KingP73] have called
this LOCK-UNLOCK and we shall use this latter terminology. Since a LOCK may be requested for a data element while other elements have been previously LOCK'ed, we can have wait-for.

(4) Circular wait. A user may issue a LOCK for a data element which has been previously LOCK'ed by another user. The first user is blocked, and must wait. Meanwhile, if the second user attempts to LOCK a data element LOCK'ed by the first user, a circular wait condition exists. For any access with characteristics like those of LOCK (exclusivity and the possibility of initiating other access of the same or different type before the completion of the original access) the possibility of wait-for and circular wait exists.

Data base deadlocks are a result of contention for data in the Logical Data Base.

How can we use deadlocks to provide protection? From the definition of a deadlock-based mechanism, we want to create a system in which any access attempt denied under the protection pattern will create a deadlock. The system could then at a later time look for deadlocks, knowing that the user who attempted the illegal access would be held in
"suspended animation" until the system broke up the deadlock. By stipulating the existence of the LOCK access, we have ensured the possibility of deadlocks. To have a deadlock-based mechanism we posit the existence of a process in the system which has LOCK'ed some of the system's logical data resources and which is itself blocked waiting for something that some user process has. If the user process makes a request for something LOCK'ed by the posited process, a deadlock will occur, due to the closure of a circular wait condition.

It is clearly not reasonable to consider that any user process would "lay down its life" by deadlockng with an attempted penetrator. Rather this posited process must be of a special nature.

Define: A pro-process is the means by which a deadlock-based mechanism causes deadlocks with attempted penetrators. The deadlock-based mechanism employs pro-processes and their orchestrated resource requests to create a situation in which any user access attempt that is denied under protection pattern will result in a deadlock between the user process and one (or more) of the pro-processes. (The "pro" in pro-process is meant to suggest protection and
A multiple user system in which:

(a) there is one pro-process associated with each user process,
(b) each pro-process is blocked waiting for the "death" of the associated user process,
(c) for each user \( u_i \), the corresponding pro-process has \( \text{LOCK'ed} \) those elements, \( x \), of the Extended Logical Data Base for which \( U(x) = u_i \) and \( P(x) = \text{deny} \), and
(d) the pro-process resource requests are exclusive only with respect to the associated user process (i.e., other user process can gain control of a resource controlled by a non-associated pro-process),

will have a deadlock if any user process attempts an access denied under the protection pattern. Further, the only deadlocks between user processes and pro-processes are the result of attempted access that are denied under the protection pattern.

By (c) the pro-process for user \( u_i \) has \( \text{LOCK'ed} \) those resources for which \( U(x) = u_i \) and \( P(x) = \text{deny} \). From (d) the pro-process requests are
exclusive only with respect to the associated user process. This creates a situation in which the pro-processes cannot satisfy the mutual exclusion and no pre-emption conditions required for deadlock except with respect to the single user process with which the pro-process is associated. A deadlock between a user process and its associated pro-process will occur if and only if the user process completes the circular wait condition by attempting access which is denied by the protection pattern. The pro-process has LOCK'ed all resources for which \( P(x) = \text{deny} \). This LOCK'ing satisfies the mutual exclusion, wait-for and no pre-emption conditions. The pro-process is blocked waiting for the user process to "die". If the user process becomes blocked waiting for the pro-process to UNLOCK the (illegally) requested resource, a circular wait condition exists. Thus, for any illegal access we will have a deadlock.

If there is a deadlock between a pro-process and a user process it is the result of an illegal access attempt by the user
A deadlock must include a circular wait condition. The user process must have made a request for something that the pro-process has LOCK'ed. The only things that the pro-process has LOCK'ed are those resources to which the user process is denied access. Thus, the circular wait condition (and the deadlock itself) is the result of an access attempt for some resources to which access is denied.

This theorem provides the necessary and sufficient conditions for what we shall term security deadlocks, deadlocks which result from an illegal access attempt and involve only the offending user process and the associated pro-process. There remains the possibility of deadlocks resulting from contention for system resources or from improper use of process synchronization mechanisms. We shall term these basic deadlocks. A third type of deadlock can potentially exist. If a user process LOCK's some resources, then attempts an illegal access, a second user process with legitimate access to the LOCK'ed resources will find itself blocked (and thus deadlocked) if this second user process attempts to exercise those legitimate access rights. We shall call these situations mixed deadlocks.
The deadlock-based mechanism operates by having a pro-process \texttt{LOCK} the resources to which the corresponding user process is not permitted access. If the user attempts an access which violates the protection pattern, a deadlock ensues. The hidden question here is how the system determines whether an access attempt is legal or not. What we must ask is how a resource request is made and how it is handled by the system.

Consider a deadlock-based protection mechanism which operates as follows: for each process (pro-process and user process) the system maintains a \texttt{LOCK} list, containing those resources which have been \texttt{LOCK}ed by other users. When a user makes a request, it is checked against this list. If the requested resource has been \texttt{LOCK}ed by another process, the requesting user is blocked and is placed on a wake-up queue for the requested resource. This queue is scanned when the original \texttt{LOCK}ing process \texttt{UNLOCK}s the resource, the previously blocked processes being allowed to contend for the now \texttt{UNLOCK}ed resource. Clearly this system will have deadlocks whenever a circular wait condition exists, and thus, with proper pro-process resource requests, can operate as a deadlock-based protection mechanism. We can see, however, that the \texttt{LOCK} lists are "inverse" capability lists in the sense of the earlier reference to Dennis and
Van Horn [DennJ66]. They contain the capabilities that a user may not exercise.

Another choice is to distribute the information about LOCK'ed resources over the database. Such a system would operate in the following manner. We view the database as has been previously discussed. Each record is a set of (attribute, value) pairs and pointers to other records based on shared keywords. Further, each record contains a LOCK list, indicating which processes may retrieve the node. Each attempted access would require checking the LOCK list to see whether the process could access the record. (A variation is to associate a LOCK list with each pointer, indicating which processes may retrieve the "pointed to" node). Again, deadlocks can exist due to circular wait conditions. This implementation results in an access list system like that used in MULTICS described by Saltzer [SaltJ74]. The record LOCK lists are "inverse" access lists in the same sense as the "inverse" capability lists of the previous example.

As a final example, consider a system which performs some algorithm on each request to determine whether or not a deadlock would ensue if the request were granted. Such an algorithm would have coded into it the access control information. This type of system realization results in a
formulary-like system like that described by Hoffman [HoffL71].

In all three examples a particular access is denied by having a pro-process request made for that access. The different systems result from how deadlocks are detected. One should note that detection of security deadlocks is a much simpler problem than detecting basic or mixed deadlocks. This is because there is only one process (the associated pro-process) that must be examined to see whether a deadlock exists. In fact since we know a priori that if the pro-process has LOCK-ed the resource, then it is reasonable to check in advance of attempt to grant the access to see whether the pro-process "has" the resource.

Since deadlock-based protection mechanism are actually implemented as more conventional (i.e., access list, capability list, etc.) structures, why introduce deadlock based systems at all? The first reason is that we are able to utilize a single, very familiar mechanism to explain a variety of different protection structures. Second, such a structure identifies exactly what must be done to prevent a given access from being completed. This model also is somewhat more exact than the "unforgettable token" models proposed by some authors [JoneA73, WulfW74]. In these other systems a process must have such a token to access the
object associated with the token. Such models leave open
the question of how this is to be done. We state this
explicitly and exactly in the theorem above. Finally the
deadlock model focuses precisely on the most basic function
of any protection mechanism, preventing the completion of
illegal accesses.

Summary

In this chapter we have presented a model for the data
base structure and the system structure. We have suggested
a formal record partitioning scheme which is shown to
possess the essential characteristics for a Data Secure
System. The compartmentalization induced by the
partitioning and the disjoint list property means that
access control decisions need only be made for clusters
rather than for individual records. If \( X \), the basis for the
logical equivalence relation, contains all the keywords of
the system, then the LOG equivalence classes are shown to be
atoms of the Boolean algebra of retrievable sets of records.
We have also shown that the creation and maintenance of a
data base utilizing these principles requires only
reasonable amounts of effort. In the next chapter we shall
consider in detail the algorithms for record insertion and
access. We shall also consider the cases in which \( X \) does
not contain all of the keywords in the system.
Chapter IV. The Engineering Model.

Introduction

Much of the preceding development has been largely theoretical in character. It is the goal of this segment of the dissertation to relate these theoretical considerations to the practical realities of system design. Our hope is to provide guidance to the Data Secure System designer in the form of detailed algorithms and examples.

The Model

We have discussed the system in the preceding paragraphs from the viewpoint of somewhat disjoint problems and their solutions. Our goal in this section is to attempt to put the system in better perspective, considering it in its entirety by presenting an engineering model. Since this model is a framework within which the implementation can be performed, we shall tend to dwell upon those features which are least obvious and which thus offer the most challenge to the system designer.
In our treatment of the pass through problem we suggested the use of the atoms of the Boolean Algebra of queries as a solution (i.e., the logical equivalence classes when \( X \) contains all the keywords of the file). While the unrestricted use of this idea is quite attractive from a theoretical standpoint, in a practical context we must temper its use.

Consider a file in which each record has at least one unique keyword, for example, accession numbers in a document system. Then every record is in a different atom and each atom has exactly one record in it.

Actually, the problem is more general than this unique keyword example. In most (if not all) practical systems there are certain keywords which may be a priori excluded from consideration when discussing potential protection specifications. These represent "don't care" conditions. That is, those keywords are irrelevant to the security structure. Thus, we need only extract atoms of the remaining sub-algebra. More formally, we have the following definitions.

**Defn:** A security keyword is an (attribute, value) pair which may be used in a protection specification. Only security keywords may be so used.

**Defn:** The security algebra is the Boolean algebra of
sets of records which are retrieved in response to queries made up only of security keywords.

**Defn**: A security atom is a collection of records that is an atom of the security algebra.

Clearly, every security atom is made up of the union of one or more atoms of the query algebra. (Query atoms, to distinguish the two types. In the sequel when we refer to atoms in an unqualified way, we shall mean security atoms.) All of the previous development of the atom concept still applies. What we are saying is that there are certain keywords whose presence or absence can never have any effect on the protection of a given record. From a security standpoint the non-security keywords are irrelevant and need not be considered.

The security algebra denoted by $B(Q')$ is a formal characterization of the notion of a security structure different from the nominal system structure. $B(Q')$ has a more "grainy" structure than the query algebra $B(Q)$.

To avoid a potential problem, we further stipulate that within a given record no two security keywords have the same attribute. By making this stipulation, we avoid the problem of conflicting security requirements induced by ambiguity in the description of the record. We also drastically reduce the
number of security atoms, as we shall show later.

A question should now be posed: How do we choose keywords for inclusion as security keywords? Naively, we might suggest simply making a yes/no decision for each keyword in the system. This, however, is impossible in a large system having many keywords. Again we seek some way to deal with aggregates of keywords. By far the most reasonable way to choose security keywords is to utilize all those keywords which have some attribute, or one of a set of attributes, regardless of what particular value that attribute might assume. This matches our intuitive ideas, in that we know in advance that some attributes, e.g., military security classification, are definitely important for protection no matter what the value is, and that other attributes, e.g., accession numbers, are not important for protection.

**Defn:** A security attribute is the attribute of a security keyword.

We suggest that the security keywords are in fact selected because they have a security attribute as their attribute.

It is in this area where we may employ the ideas of logical equivalence and clusters in a more general way. We introduced the logical equivalence relation by stipulating that a set of keywords, X, had been specified. The
membership of a record in a logical equivalence class was determined by whether or not the record satisfied the canonical expression of the keywords of \( X \) associated with the equivalence class. Thus, if \( X \) contains all the security keywords, then the records in a given logical equivalence class will be afforded uniform protection under any protection pattern. This occurs because the Boolean expression of any TYPE.5 protection specification can be decomposed into minterm (canonical form) expressions of the keywords from \( X \). All the records within a logical equivalence class satisfy one and only one of these expressions. Therefore, the original protection specification has the effect of permitting or denying access to logical equivalence classes. When the records are inserted into the data base (and thus assigned to cell and structural equivalence classes) we have required that the no cluster include members of different logical equivalence classes. So, our original TYPE.5 specification has the net effect of permitting or denying access to a cluster. If \( X \) contains only the security keywords, the logical equivalence classes represent atoms of the Boolean Algebra of all "protectable" sets of records, which we have called the security algebra. By this we mean the sets of records which would be retrieved in response to the allowable "queries" of TYPE.5 specifications.
It should be noted again that by separating the logical equivalence relation from the cell and structural equivalence relations, we have allowed for more flexibility in the data base model than previous treatments of similar models. All the member records of a cluster are "close to each other" in a physical sense due to their being in the same cell. They are also "close" in a logical sense because they satisfy the same Boolean expression of keywords. All the records of a cluster are linked by the file structure, so these closeness properties may be utilized in retrieval and update processing.

We have deliberately avoided introduction of a security equivalence relation because in this model the keywords are the only characteristics of records which may be used for retrieval specificatons. Thus, the logical equivalence relation can be employed to partition the records into security classes. If non-keyword (attribute, value) pairs are deemed significant to the protection of a record, two choices exist for accomodation into this model. The first is simply to designate the significant pair a keyword. This is quite reasonable since our definition of a keyword was quite loose. If, however, this is not possible, a second choice would be to extract the relevant information and create a keyword. This is precisely what is done in a
military context. The content of a document is analyzed against pre-defined standards and a keyword such as (Classification, Top Secret — No Foreign Dissemination) is assigned. Under this latter artifice, much of the work in automatic document analysis may be utilized in this model. By the use of either of these two techniques, we are able to avoid introducing a special-purpose security equivalence relation. This is in keeping with one of our basic goals in modelling Data Secure Systems, that the security considerations be a fundamental and integral part of the system.

Before we dispense with these points, let us illustrate them with a brief example.

Example: Query Atoms and Security Atoms

Consider the following six records. The reader can verify for himself that there are six query atoms, each with one member record. Furthermore, if every unique keyword is chosen as the security keyword, then there are six security atoms.
Using these records, let us suppose that we have determined that the attribute, "subject", is a security attribute. Then we have the following potential security atoms, each corresponding to a canonical expression of security keywords.

Figure IV.1 - Records for atom example.
Figure IV.2 - Security atoms.

This example illustrates several important factors. First, we have gone from six query atoms to only two security atoms by stipulating that only subject is relevant for security purposes. Secondly, half of the potential security atoms are empty. We can get some rough information on how many of the potential atoms are likely to be empty. Suppose that we have selected \( n \) different attributes as security attributes. Further, let the security attribute \( i \) have \( v[i] \) distinct values. The number of security keywords is:

\[
x = \sum_{i=1}^{n} v[i]
\]

There are \( 2^x \) possible logical equivalence classes (security atoms). However, the definition of a security keyword stated that no two security keywords in a record have the same attribute. Thus we have only...
potential atoms, since we can choose from \( v[i] \) different values for each of the \( n \) different attributes. This occurs because many canonical form expressions cannot represent atoms since the expressions violate the requirement that no two security keywords in a record have the same attribute. It is obvious that

\[
y < < 2^{**x}.
\]

Consider the following where we let:\n
\[
\]

\[
x = \prod_{i=1}^{4} v[i] = 18 \quad \text{and} \quad 2^{**x} = 262,144.
\]

Also

\[
y = \prod_{i=1}^{4} v[i] = 360.
\]

Thus, \( y < < 2^{**x} \).

Finally, it may be possible to further refine our estimates on the maximum number of non-empty potential security atoms. If no record can have more than \( m \) security keywords, the actual number of atoms is at most the product of the \( m \) largest \( v[i] \)'s.

Hsiao [HsiaD75] has correctly pointed out that consideration of data base structure is closely linked to
consideration of the associated data base system algorithms. In this light, let us present the basic algorithms for search and maintenance of the data base. We begin by defining some operators.

**Defn:** Cluster identifier operator, \( b(K) = \{ c[i] \} \) where 
\[
(c[i], a[i], t[i]) \in E(K), i = 1, \ldots, h.
\]
Simply, the operator returns the cluster identifiers for all the clusters that include records for which keyword \( K \) is true.

**Defn:** Record count operator, \( n(K,c) = t[i] \) where 
\[
(c[i], a[i], t[i]) \in E(K) \text{ and } c[i] = c.
\]
This operator gives a count of the records in cluster \( c \) which have keyword \( K \) true, i.e. the number of records on the \( K \)-list in cluster \( c \).

**Defn:** Head of list operator, \( r(K,c) = a[i] \) where 
\[
(c[i], a[i], t[i]) \in E(K) \text{ and } c[i] = c.
\]
The operator returns the addresses of the first record in the \( K \)-list in cluster \( c \).

**Defn:** File search operator, \( s(K,a) \) is the record address that is the \( K \)-pointer for the record whose address is \( a \). (If the \( K \)-pointer is null, \( s(K,a) \) will yield 0).

As thus defined, \( s(K,a) \) is a formal description of the notion of tracing through the file via the \( K \)-pointers. The function returns the address to the next record on the
K-list. We shall say that a record R has been accessed or retrieved if s has been applied to produce a K-pointer of R.

Let us start with the description of a record insertion algorithm.

Algorithm: Record insertion.

Input: A record to be inserted. That is, a collection of (attribute, value) pairs, some of which are keywords and further, some of the keywords are contained in X.

Step 1: Determine which of the keywords of the record are in the set X. These keywords will determine the logical equivalence class to which the record belongs. If there are none, add to the record the artificial keyword "UNCLASSIFIED" and pass this keyword to step 2.

Step 2: Determine candidate clusters for insertion by intersecting together the sets of cluster identifiers associated with each of the keywords found in Step 1. Call the resulting set of cluster identifiers THETA.

\[ \text{THETA} \leftarrow \bigcap_{i} b(Ki) \]

i runs through all the keywords of the record that were found in Step 1.

Step 3: Delete from THETA those clusters into which the
record cannot be inserted either because the cell in which the cluster is located is full or because insertion in the cluster is not allowed by the structural relation.

Step 4: Examine the remaining clusters represented in THETA against the added record, deleting any cluster which represents a different logical equivalence class.

Step 5: If THETA is now empty, add to THETA the address of the location where a new cluster may be started. This new cluster location should be selected so that all the members of a logical equivalence class are in as few cells as possible.

Step 6: Arbitrarily select any remaining member of THETA, insert the record in that cluster, link it onto all the appropriate Ki-lists and update the directory.

Because this algorithm is fairly complex, let us attempt to motivate what is happening in it. In Step 1 we begin by determining which of the record's keywords determine the LOG class to which the new record belongs. In
the event that there are no keywords from X in the new record, we insert the artificial keyword "UNCLASSIFIED" to retain uniformity. We have assumed that we know which of the (attribute, value) pairs are keywords. Steps 2 and 4 are determining which clusters represent the same LOG class as the new record. We have deliberately formulated the insertion algorithm in this way to emphasize its similarity with the parallel search algorithm which follows immediately. If it were desired not to use this similarity, a single step could replace these two. This alternative would likely use a canonical expression-to-cluster mapping. Step 3 is also included to emphasize the similarity. In this step we delete those clusters to which insert access is denied (because they are full), just as in the search algorithm we, at the same point, delete the clusters to which the user is denied search access. Note that there are actually two types of full conditions. The first is a physical full condition; the cell in which the cluster is located does not have enough space in it to store the record. The second is a logical full condition; the STR relation will not allow more records to be inserted into the cluster. Although our examples of the logical full condition have used inverted files, STR may be used to control cluster size at values larger than one record. In this way we can achieve partially inverted files.
In Step 5 we may have to supply the address where a new cluster may be established. We have suggested that this new cluster address be chosen so that all the clusters representing a given LOG class be located in as few cells as possible. Rothnie and Lozano (RothJ74) give some excellent motivation why this is a good practice. Our work is a major generalization of theirs because we place no restrictions on the file structure and because we can offer a strong mathematical basis for the concepts. It may be possible to further optimize the physical placement of clusters, but one rapidly gets into computationally difficult areas.

We did not give a detailed record stowing algorithm in Step 6 because so many choices exist. For completeness we now present one. The reader is cautioned that other mechanizations of Step 6 are possible. As we enter Step 6 we know that the record is to be inserted into cluster c. Further, we know (or will calculate) that the record will be stored at address a. The following algorithm determines the k-pointers, updates the directory and stows the record.

When one of the previously defined operators appears on the left of an assignment statement, we meant that the directory value represented by the operator is replaced with the result of the right side of the assignment.)
BEGIN
  FOR $K_i =$ (each of the keywords of the new record) DO
    BEGIN
      new record's $K$ pointer := $r(K_i, c)$;
      // set the $K$-pointer to the current head of list address
      $r(K_i, c) := a$;
      new head of list address is address where record is inserted;
      $n(K_i, c) := n(K_i, c) + 1$;
      // increment length of $K$-list
      END;
      update first-free (or last-used) address for cluster $c$;
      store record;
    END;

This stowing algorithm has the following desirable properties. First, no previously stored records are modified. Second, it is not necessary to follow down any keyword lists to find what the pointers of the new record should be. Third, all the $k$-lists have increasing addresses as one traces down them. Directory update is simple and easy to understand. The directory update works correctly even if there was no $K_i$-list in the cluster previously. If there was no such list, the first reference to $r(K_i, c)$ would return a null (zero) value. At this point the system should include in the directory entry for $K_i$, $E(K_i)$, the triple $(c, 0, 0)$ if the was no previous entry in $E(K_i)$ for cluster $c$. When this is done, the algorithm proceeds to correctly update the directory.

We now present the companion parallel search algorithm.
Algorithm: Parallel Search

Input: A query as a disjunctive form (df) expression of keywords, where \( K[I,J] \) is the \( J \)-th keyword in the \( I \)-th term:

\[
K[1,1]\&K[1,2]\&\ldots \lor K[2,1]\&K[2,2]\&\ldots \lor \ldots
\]

Step 1: Determine candidate clusters for each term by intersecting together the set of cluster identifiers associated with each keyword in the term. Call the set of candidate clusters for the \( I \)-th term \( \Theta_{I} \).

\[
\Theta_{I} := \bigcap_{J} b(K[I,J])
\]

\( J \) runs through the keywords in term \( I \).

Step 2: Delete the denied clusters from each \( \Theta_{I} \). The information on which clusters are denied can be obtained by processing the protection specifications against the cluster descriptions to produce the protection pattern.

Step 3: Find the prime keywords. Find the keyword \( K_{\text{prime}} \) in the \( I \)-th term which has the shortest \( K_{\text{prime}} \)-list for each of the clusters in \( \Theta_{I} \) (i.e., the minimum \( n(K, \Theta_{I}) \) will cause \( K \) to become the prime keyword for this term and cluster). Merge a pair \((K_{\text{prime}},c)\) onto a list \( \Gamma_{\text{gamma}} \), where \( c \) is the cluster in which \( K_{\text{prime}} \) is a prime keyword. Repeat for each term.
Step 4: Decode directory and initialize the search list. For each (KPRIME, c) pair on \( \Gamma \), merge the beginning address of the KPRIME-list in cluster \( c \), \( r(KPRIME, c) \) onto an ascending sorted list \( \Sigma \).

Step 5: Retrieve records.

a. Find the smallest unused address, \( a^f \), on \( \Sigma \). (If all addresses have been used, exit.) Mark that address as used. Retrieve the record at address \( a^f \).

b. For each of the keywords, \( K \), that are prime in this cluster, and are present in the retrieved record, merge their non-null K-pointers, \( s(K, a^f) \), onto \( \Sigma \).

c. Process the retrieved record against the user's query. If the record satisfies the query, give the record to the user.

d. Continue with step 5a.

Again let us attempt to motivate what the algorithm is doing. The overall goal of the parallel search algorithm is to examine as few records as possible in the process of finding all the records which satisfy the query. Step 1 performs the intersection characteristic of inverted (or semi-inverted) file processing. In it we are able to
systematically exclude from consideration those clusters which cannot contain records which satisfy a given term of the query. In order for a cluster to contain any records which satisfy a term, there must be a k-list in the cluster for each of the keywords of the term. Only the records where each of these lists intersect satisfy the term. In Step 2 we delete those clusters to which search access is denied. This means that very little processing is done for a cluster which turns out to be denied. In practice these two steps can be advantageously combined. Step 3 further minimizes the number of records that are examined. Since all the k-lists for the keywords of a term must intersect at a record if that record satisfies the term, we need only follow down the shortest list. We will encounter every record which satisfies the term by following the prime keyword list, and we will minimize the number of records examined. Because a given cluster may be a candidate cluster for multiple terms, we may find different prime keywords from each of these terms for that cluster. Thus, we can have multiple prime keywords for a cluster.

This algorithm is termed the parallel search algorithm for two reasons. First, within each cluster, all the prime keyword lists are searched in parallel. Step 5b extracts the pointers of all the prime keywords from the retrieved
record and merges them onto SIGMA, which has been previously
(Step 4) initialized to contain the beginning address of
each prime keyword list for the cluster. The most important
result of this parallel list processing is that a record is
retrieved once and only once, even if it is on several prime
keyword lists.

A second kind of parallelism occurs because the
processing of records in one cluster is independent of that
in any other cluster. This fact is a direct result of the
compartamentalization induced by the disjoint list property.
If the physical I/O system and the physical representation
of the file will allow actual parallelism, as most modern
systems do, then several clusters may be processed at one
time. Additional benefits may accrue because the way in
which SIGMA is maintained and utilized and the way in which
we suggest that records are stowed, the file tends to be
traversed in a uniform direction. The record stowing
algorithm given causes the addresses of records on each
K-list to be in an increasing order. Thus, if care is taken
with the physical placement of records, this algorithm
exhibits some of the characteristics of the well known SCAN
disk algorithm [PinkT72].

Another point to note is in the area of actual query
processing. It is possible to greatly improve the
processing of a record against a query by taking advantage of the fact that all the records in a cluster satisfy the same canonical expression of keywords from X. One may be able to determine a priori that certain terms of the query cannot be (or will always be) satisfied by records from the cluster currently being examined. Further, only those query keywords not in X need be examined during the processing within one cluster. Thus, any query composed only of keywords from X will be answered with absolute precision[\textit{\textbf{[*]}}], i.e. every record retrieved will satisfy the query. Finally, since the same keywords from X are present in every record, they are the least likely to be chosen as prime keywords.

While we are on the subject of query processing, it should be noted that the disjunctive form query is probably an unreasonable restriction. What could be done is to create a front-end processor which would provide a better user interface and which would transform the queries into

\[ \text{[\textit{\textbf{[*]}} Precision is defined as:} \]

\[ \frac{\text{number of records retrieved that satisfy the query}}{\text{number of records retrieved}} \]
disjunctive form.

This approach partitions the set of records in a file into clusters, such that all the records of a given cluster have certain logical characteristics in common. Given the assumption that the first access to a cell is more costly than immediately subsequent accesses, it is a reasonable objective of a search algorithm to minimize the number of different cells assessed in retrieving records satisfying some Boolean expression of keywords. It is also desirable to do the processing in one cell before we move on to another cell. To attain these objectives, three conditions must be satisfied:

(1) Records must be assigned to cells in such a way that a small number of cells will contain records satisfying the Boolean expression.
(2) There must exist some mechanism for determining which cells contain these records without accessing other cells.
(3) The disjoint list property must be preserved.

Since all the members of a cluster are stored in a minimum number of cells, and information about this mapping is maintained in the directory, then the first two of these conditions are fulfilled. The insertion algorithm preserves
the disjoint list property. Thus, this file structure model potentially offers good efficiency and good protection.

It is difficult to discuss the efficiencies of an algorithm without detailed consideration of the precise implementation of the algorithm. Nevertheless, we can make some generalizations about the amount of "work" devoted to protection. In many previous systems, each retrieved record is processed against a description of the permitted (or denied) records of the file. Depending upon the outcome of this processing, the record is either released for further evaluation against the user's actual query or is discarded because the user is denied access to the record. In this latter case, all the effort expended to retrieve this record is wasted giving poor precision of protection[**]. In systems like Hsiao's [HsiaD68a, HsiaD68b] the precision of protection will be less than one.

Recently, more attention has been devoted to systems which systematically modify the user's query [StonM74]. Our model, in effect, does this also. In these systems a

[**] Precision of protection is defined as:

\[
\text{Precision of protection} = \frac{\text{number of records retrieved but later denied}}{\text{number of records retrieved}}
\]
description of the permitted portions of the file is "anded on" to the user's query. While such systems typically have protection precision of unity, the use of such a technique may incur substantial penalties in directory processing if a user with broad access privileges makes a relatively narrow query. The query is enlarged to include descriptions of all the permitted sections of the database. In our model, query modification is postponed until after preliminary processing of the query. Thus, access control decisions are made for only those clusters from which the query would actually attempt to extract records. Another major point is that we are able to make our access control decisions for clusters, not individual records, thereby making fewer access control decisions.

As a final point before we discuss some extensions of the parallel access algorithm, let us give an argument about the correctness and completeness of the above formulation. The insertion algorithm assures that a record cannot be inserted into a cluster representing a different logical equivalence class (Step 4 of the insertion algorithm). Therefore, a record cannot be misdirected and accidentally downgraded. We shall assume Step 6 of the insertion algorithm is written to make sure that the disjoint list property is preserved, since this is an easy thing to check.
The only way that a record can be retrieved by the search algorithm is for its address to be found on SIGMA. Addresses are placed on SIGMA in two ways, either by being the K-pointer for a prime keyword or by being the head of list address for a prime keyword list. From the disjoint list property, if a K-pointer is inserted into SIGMA, the head of that list in that cluster must have been originally inserted onto SIGMA by Step 4 of parallel search. Thus the pair \((K,c)\) must have been on GAMMA, where \(c\) is the cluster under discussion. In order for this to have occurred, \(c\) must have been a candidate cluster for some term, and \(K\) must be the prime keyword of that term for cluster \(c\). But in Step 2 of parallel search we deleted all clusters to which the user was denied access. Thus, we would not have attempted to find prime keywords, and no \((K,c)\) pair could have been inserted onto GAMMA. Thus, no head of list addresses could have been inserted onto SIGMA for lists in denied cells. SIGMA therefore is initialized with addresses only from permitted clusters. Since no K-pointer in a permitted cluster can point into a denied cluster (by the disjoint list property), no addresses of denied records will ever appear on SIGMA, and no denied records will ever be retrieved.
Extensions of the Basic Algorithms

Although it handles only one query, i.e., one disjunctive form expression of keywords at a time, the access algorithm can be easily extended to handle queries from multiple users. The extension requires the addition of a query queue, the formation of multiple queries into a (combined) disjunctive form, and the dispatch of records to all the users whose query is satisfied by the record. The queue is used to stack up the queries. Each of the queued queries will be transformed into a disjunctive form. At the end of a waiting period, a combined DF is then formed:

$$ DF[1] \lor DF[2] \lor \ldots \lor DF[n] $$

In this case, there are $n$ users and $DF[i]$ denotes the disjunctive form of the query from user $i$. Obviously, the disjunction of disjunctive forms is again a disjunctive form. It is this combined DF that will be used by the parallel access algorithm as input.

Two modifications to the basic parallel access algorithm must be made to handle these combined queries. Both are necessitated by the fact that different users will likely have different access rights (e.g., permit or deny) to the same clusters. The first modification consists of processing the query of user $i$ through Steps 1, 2, 3 and 4.
before starting on the query of another user. In the terms of the algorithm we have:

Step 0: FOR I=1 STEP 1 UNTIL (number of users) DO
BEGIN
Step 1;
Step 2;
Step 3;
Step 4;
END;
Again these steps communicate with the retrieval step only via the initialization of SIGMA, the list of records to be retrieved. The deletion of denied clusters in Step 2 is, of course, done on the basis of the particular user whose query is currently being processed. Notice that no modification of these four individual steps was necessary to accommodate multiple users.

The other modification required for the multiple user case is in the actual retrieval, Step 5. Because we do not wish to impede the searches of highly privileged users, we must develop a way for them to access clusters to which other "concurrently batched" users are denied access. We shall call this the "awake/asleep" mechanism. Fundamentally, we "put to sleep" users who are denied access to a cluster when we are about to retrieve records from that cluster in response to other authorized users. Whenever a new cluster is about to be accessed, we update the awake/asleep status of each user. When a user is in the
"asleep" state, no records may be dispatched to that user. The awake/asleep mechanism insures that even records which satisfy a user's query will not be released to him if they were retrieved from a denied cluster.

So our revised retrieval step (Step 5 of the search algorithm) becomes:

Step 5*: Retrieve records.

WHILE (SIGMA is not empty) DO
BEGIN
a* ADDRESS*:=(smallest unused address on SIGMA); 
IF (ADDRESS is from a new cluster) THEN
BEGIN
   update current cluster;
   update Awake/Asleep information;
END;
Get record at ADDRESS;
b* FOR KWD*=(each keyword prime in this cluster) DO
   Merge s(KWD, ADDRESS) onto SIGMA;
   FOR I*=(each awake user) DO
      IF (retrieved record satisfies DF[I]) THEN
         Give record to USER[I];
END;

We note that the single user case did not require this modification because no denied clusters were processed in Steps 4 and 5, hence no denied addresses were placed in SIGMA initially. From the disjoint list property, that all the records on any given K-list are in a single cluster, no addresses from other (possibly denied) clusters could be added to SIGMA in the course of the algorithm. Hence, in the single user case, we could use a less elaborate retrieval step.
Again, the processing in one cluster is independent of that in any other cluster, so in an implementation, it might be possible to utilize the parallel characteristics of most modern I/O subsystems. In such an implementation it may also be desirable to organize the algorithm in different ways to achieve various efficiencies.

Multiple Access Types

When we introduced the idea of multiple access types in the previous chapter, it was achieved by the potential existence of procedures associated with the records. These procedures would be automatically invoked when the record was accessed, and would have only the pointer information of the record in the procedure's environment. It is possible to distinguish between these procedures on the basis of whether or not the access type can be reached via the normal keywords of the file. Figures IV.3 and IV.4 are an attempt to show these two cases.

In the Figure IV.3, there is a hierarchical file and access type structure. First the user must specify the access type then supply keywords to the system for processing of the file directories at the next lower level. In the figure we also note that access types A and B share a
single file directory while access type C has a separate directory. In the Figure IV.4 any reference to cluster 0 causes the procedure for Access Type D to be invoked. Access to clusters 1 or 2 cause Access Type E to be invoked, hence the double line from the directory through Type E. Such invocations may be invisible to the user. Accesses to clusters 3 and 4 have no such procedures. Of course these two basic cases can be combined to give more elaborate access types.

Figure IV.3 - Access types for entire file.
Figure IV.4 - Access types for subfiles.

Other Data Base Operations

So far we have considered two main file operations, the insertion of records, and their retrieval in response to queries. These two operations were considered together because they have the most impact on the security of a database system. Errors and omissions in either design or implementation of these two operations have the most obvious implications for security. These two algorithms (as the careful reader has surely observed) are quite similar in the steps which do directory processing. In the insertion algorithm we, in effect, make a query of the keywords from the inserted record that are contained in X. The access type of "insert" is denied to target clusters which are
full, (Step 3 of the insertion algorithm), in the same way as the access type of "search" is denied (Step 2 of the search algorithm) for clusters to which the user is denied such access.

There are other activities of the system which must be included for completeness. The first of these is deletion.

Deletion of a record is actually a two stage process. First we mark the record as one to be deleted. Then, at a possibly later time, we actually do the deletion by bridging over the pointers and freeing the space. We may wish to postpone the actual deletion until a later time to allow for a single garbage collection routine to be periodically invoked. The garbage collection in one cluster is, of course, independent of that in any other cluster. One should realize that deletion of a record has no significant impact on the security structure of the system. It would seem prudent to limit the access type of "delete" to those records to which the user has search access, and always to return an indication that the record has been deleted, even if the user was denied search access to the record (and thus could not really delete it). This is the tip of an iceberg of problems relating to the transmission of information by denial of access. We shall give some indication of the kind of work which needs to be done in this area in the
Let us turn our attention to another data base activity, the maintenance of the protection specifications. It has been argued that protection of the protection information itself should be naturally included in the same scheme that allows for protection of the rest of the information in the file [HsiaD68a, GrahG72]. In this light it is desirable to have this protection information be manipulated in the same way that the files are manipulated. If one adopts such a view, it is possible to economize on the system routines because the data base maintenance routines for the file of protection specifications are exactly those for the rest of the system files.

How will this protection specification file be used? With the answer to this question we can then suggest how it should be structured. The major use of the file is in retrieving whether access to a cluster is to be permitted or denied based on a user's request. At once we can see the possibility for a record of four (attribute, value) pairs, as shown in Figure IV.5.
We note that this record corresponds closely to our conceptual model protection patterns. The difference being that we have replaced the data identifier of the protection pattern with the cluster identifier which characterizes a possibly large number of individual records all of which have uniform protection due to their membership in the same logical equivalence class. This cluster identifier is determined through processing the TYPE.5 protection specifications against the logical descriptions of the clusters.

A choice of record structure like that of Figure IV.5 does have the disadvantage of requiring a record retrieval for each cluster to be examined. It would be desirable then to have an organization which minimized the number of retrievals required to get the necessary protection information. In the multiple user parallel access algorithm this same information must be used twice. The first time is
when the clusters denied for this user and access type are deleted from the list of candidate clusters. The second time is when a new cluster is about to be opened for retrievals. Here the awake/asleep status of the users must be updated in accordance with their access rights to the newly opened cluster. Thus it seems most reasonable to suggest that the records of the protection pattern be associated with a single user. Further, in the interests of efficiency, we should actually retrieve the protection record only once per query, retaining it for subsequent use in the two different places. This means that the protection records are somewhat like capability lists [DennJ66]. However, there are many situations in which we wish the protection information were otherwise organized. For example, consider the difficulty of altering the security classification of a cluster. In this simple structure it would be necessary to examine the protection record of every user who might have access rights to the file, even though many users did not really have such access rights. This type of update situation is best handled by an access list structure [SaltJ74]. We can achieve this with our structure by simply having the records for a given cluster threaded on a list. Thus the structure of a protection record becomes like that shown in Figure IV.6.
(USER.ID, u), a pointer to another record with same user
(ACCESS.TYPE, a") a pointer to another record with same access type
(CLUSTER.1, c[1]), a pointer to another record with same cluster
(CLUSTER.2, c[2]), a pointer to another record with same cluster.

Figure IV.6 - Improved protection record.

With the inclusion of pointers for the user and access type, other updating situations can also be handled with facility. This organization is very similar to the one of authority items [HsiaD68]. We have deliberately left out the permit/deny indicator. It is desirable to have multiple, independent checks on security critical decisions [FabR73]. One could thus maintain one record which described the permitted clusters and another for the denied clusters as suggested by Manola in [ManoF71]. In order for a request to succeed in accessing a cluster, the cluster would have to be permitted and also not be denied. The protection records shown above are actually created by processing the protection specifications against the cluster descriptions.

Of course, the right to alter or to read the protection information must be carefully guarded. The disjoint list property and the compartmentalization which it induces are
of benefit here also, because this protection information will be segregated from the rest of the data base, in contrast to \([\text{FrieT70}]\), and can be afforded the appropriate controls.

**Examples**

Let us conclude this discussion of the engineering model by presenting some examples of the algorithms. For all of them, we shall utilize the records shown in Figure IV.7. The query algebra atoms have one member each, with the exception of the atoms described by:

\[
K_1 \land K_2 \land K_3 \land K_4 \land K_5 \land K_6 \quad \text{and} \quad \\
\overline{K_1 \land K_2 \land K_3 \land K_4 \land K_5 \land K_6}
\]

which have two member records.
Figure IV.7 - Records to be inserted.

Let keywords $K_4$, $K_5$, and $K_6$ be the security keywords and let $X = (K_4, K_5, K_6)$, only the security keywords. (These would actually have been chosen on the fact that they have a security attribute.) The potential logical equivalence classes are shown in Figure IV.8 (in this figure and throughout the sequel, we will not show negated keywords explicitly; further, since the non-security keywords do not affect the logical equivalence classes they are also omitted).
Several points should be noted here. First, there is one potential logical equivalence class that is empty, K4 & K5. That is, at the instant we have chosen to view the database, no records have this exact composition of security keywords. A second point of interest is that there is one record with no security keywords true. We touched upon this same point in the conceptual model when we were considering a set of protection specifications which did not cover the Extended Logical Data Base. At that time we stated that the decision to permit or deny access to such elements would have to be made on an arbitrary basis, such the principle of least privilege. Nevertheless, the system should make creators of protection specifications cognizant of the possibility of existence of such records. We also touched upon the point in Step 1 of the insertion algorithm where it was suggested that the artificial keyword "UNCLASSIFIED" be inserted into such records. This inserted keyword is not shown here. The identifiers have been arbitrarily assigned to allow simple and compact reference to the logical classes. Henceforth, we will refer to them by their identifiers.
<table>
<thead>
<tr>
<th>Security Atom Description</th>
<th>Assigned Identifier</th>
<th>Number of Member Records</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>K5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>K5 &amp; K6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>K4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>K4 &amp; K6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>K4 &amp; K5</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>K4 &amp; K5 &amp; K6</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>NO SECURITY</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure IV.8 - Potential security atoms

To see how the record insertion algorithm works, let us focus on those records which made up class 2, and see how the data base and the directory change as they are added. Records will be linked by filling the cell from the highest address to the lowest address, adding the record at the head of the K-lists, and adjusting the directory as was suggested above. This saves tracing down any K-lists, which would be necessary if we added records at the end of the lists. From Figures IV.7 and IV.8 there are going to be four records in this logical equivalence class (class 2). We have arbitrarily chosen the cell size, s, to be three records, which will illustrate the creation of two clusters. (The same illustration could have been achieved by stipulating that no cluster have more than three member records under the structural equivalence relation and allowing a large, and more realistic cell size.)
We start the example with the first of these records already in the data base, as shown in Figure IV.10.

Data Base

Part of Directory File

\[ E(K1) = \{\ldots(3,(12,1))\ldots \} \text{ CLUSTER 3} \]
\[ E(K3) = \{\ldots(3,(12,1))\ldots \} \]
\[ E(K5) = \{\ldots(3,(12,1))\ldots \} \]

Figure IV.10 - Data base with one record of class 2.

Figure IV.11 shows the directory and the file as the remaining records of logical class 2 are added. Let us examine the insertion algorithm in detail as the fourth record of class 2 is inserted.
Data Base
Part of the Directory
\[ E(K1) = \{\ldots,(3,12,1),\ldots\} \]
\[ E(K3) = \{\ldots,(3,11,2),\ldots\} \]
\[ E(K5) = \{\ldots,(3,10,3),\ldots\} \]

Insert \[ K3 \]

\[ E(K1) = \{\ldots,(3,12,1),\ldots\} \]
\[ E(K2) = \{\ldots,(3,10,1),\ldots\} \]
\[ E(K3) = \{\ldots,(3,11,2),\ldots\} \]
\[ E(K5) = \{\ldots,(3,10,3),\ldots\} \]

Insert \[ K2 \]

\[ E(K1) = \{\ldots,(3,12,1),\ldots\} \]
\[ E(K2) = \{\ldots,(3,10,1),\ldots\} \]
\[ E(K3) = \{\ldots,(3,11,2),\ldots\} \]
\[ E(K5) = \{\ldots,(3,10,3),\ldots\} \]

Insert \[ K5 \]

\[ E(K1) = \{\ldots,(3,12,1),\ldots\} \]
\[ E(K2) = \{\ldots,(3,10,1),\ldots\} \]
\[ E(K3) = \{\ldots,(3,11,2),\ldots\} \]
\[ E(K5) = \{\ldots,(3,10,3),(4,15,1),\ldots\} \]

Figure IV.11 - Record insertion sequence.
In Step 1, the keyword K5 is found to be in X. When the intersection of Step 2 is performed, THETA will contain cluster identifier 3, and possibly others from portions of the directory not shown. However cluster 3 is full, having three records in it, so Step 3 removes this cluster from THETA. Because none of the other members of THETA represent logical class 2, they would all be removed in Step 4, leaving THETA empty. Step 5 would add back into THETA the address of a new, empty cluster, cluster 4. The record is inserted there, and the directory is updated as shown in the final portion of Figure IV.11. The final structure of the file and the directory is shown in Figures IV.12 and IV.13.
Figure IV.12 - Resulting file.
E(K1) = (K1,7: (0,2,2), (1,4,1), (2,8,1), (3,12,1), (5,18,1), (6,21,1), (7,24,1))
E(K2) = (K2,5t (1,4,3), (2,7,1), (3,10,1), (5,16,2), (8,27,19))
E(K3) = (K3,6t (0,2,1), (1,5,1), (2,7,3), (3,11,2), (5,16,1), (9,30,1))
E(K4) = (K4,5t (0,1,3), (1,4,3), (5,16,3), (7,24,1), (9,30,1))
E(K5) = (K5,4t (2,7,3), (3,10,3), (4,15,1), (5,16,3))
E(K6) = (K6,5t (0,1,3), (2,7,3), (5,16,3), (6,21,1), (7,24,1))

Figure IV.13 - Directory of file.

An important point of this file structure is that we can accommodate a variety of protection patterns without reorganizing the file, in particular, a change in the protection pattern does not require reorganization. To see this, consider the file and its directory as depicted in Figure IV.12 and IV.13. We now suggest several different protection patterns. For simplicity assume that there are three users U1, U2, and U3 and that for the first two examples there is only a single access type, hence we shall omit access type in these protection specifications. We
show the protection pattern by listing the TYPE.5 protection specifications which comprise the pattern. We also assume that any access which is not explicitly denied is permitted (a violation of the principle of least privilege). The denied cells are determined by processing the TYPE.5 specifications as discussed above.

Example: Protection specifications as applied to a file.

Case 1:

Protection Specifications

TYPE.5 ((U1, U2), K4 V K6, deny)

TYPE.5 ((U3), K5 V (K4 & K6), deny)

Resulting Protection Pattern

<table>
<thead>
<tr>
<th>User</th>
<th>Denied Clusters</th>
<th>Permitted Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>1, 3, 4, 5, 6, 7, 9</td>
<td>0, 2, 8</td>
</tr>
<tr>
<td>U2</td>
<td>1, 3, 4, 5, 6, 7, 9</td>
<td>0, 2, 8</td>
</tr>
<tr>
<td>U3</td>
<td>0, 2, 3, 4, 5, 7</td>
<td>1, 6, 8, 9</td>
</tr>
</tbody>
</table>
Case 2:

Protection Specifications

TYPE.5 ((U1, U3), K4 & K5, deny)

TYPE.5 ((U2), K4, deny))

Resulting Protection Pattern

<table>
<thead>
<tr>
<th>User</th>
<th>Denied Clusters</th>
<th>Permitted Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>5</td>
<td>0,1,2,3,4,6,7,8,9</td>
</tr>
<tr>
<td>U2</td>
<td>0,1,5,7,9</td>
<td>2,3,4,6,8</td>
</tr>
<tr>
<td>U3</td>
<td>5</td>
<td>0,1,2,3,4,6,7,8,9</td>
</tr>
</tbody>
</table>

Case 3: Multiple Access Types.

Consider the protection patterns resulted from the following TYPE.5 protection specification where Q is now a Boolean expression of keywords qualified by access types A or B.

Protection Specifications

TYPE.5 ((U1), (A & K4 & K6) V (B & K5), deny)

TYPE.5 ((U2), ((A & K4) V B); deny)

TYPE.5 ((U3), (A & K5) V (B & K6), deny)
Let us now illustrate the parallel access algorithm using this file and the protection pattern defined in case 3 above. Note that the file has the structure shown in Figure IV.14.

![Diagram of file and access type structure](image-url)

Figure IV.14 - File and access type structure for example
Let us consider the following query:

From U1: A & K2
From U2: A & K2 & K5
From U3: B & K1 & K4

Resulting Query: (A & K2) V (A & K2 & K5) V (B & K1 & K4)

We have limited the queries to a single conjunct each, to make the example a bit more tractable. Let us now follow through the algorithm.

Step 1: Determine Candidate Clusters for the Query

THETA [1] is {1,2,3,5,8}
THETA [2] is {2,3,5}
THETA [3] is {0,1,5,7}

Step 2: Remove Denied Clusters

After deletion of the denied clusters:

THETA [1] is {1,2,3,8}
THETA [2] is {2,3}
THETA [3] is {1}

Step 3: Determine the Prime Keywords from the Terms

We have:

from THETA [1]: K2 for clusters 1, 2, 3, and 8
From THETA [2]: K2 for clusters 2 and 3
from \( \Theta \) : K1 for cluster 1.

So the \( \Gamma \) list is

\[
\begin{align*}
(K1, 1) \\
(K2, 1) \\
(K2, 2) \\
(K2, 3) \\
(K2, 8)
\end{align*}
\]

Step 4: Initialize \( \Sigma \) by finding the head-of-list addresses for each member of \( \Gamma \).

\( \Sigma \) becomes \((4, 7, 10, 27)\).

Step 5: Record Retrieval

a. Since the lowest address in \( \Sigma \), 4, takes us into a new cluster, we must update the awake/asleep information.

Awake becomes U1 and U3

Asleep becomes U2

b. Get the record at 4:

\[
\begin{array}{c|c}
K1 & 0 \\
K2 & 5 \\
K4 & 5
\end{array}
\]

c. Update \( \Sigma \) to include K-pointers of all keywords prime for cluster 1, K1 and K2.
Only K2 has a non-null pointer, 5, so we merge 5 into SIGMA.

SIGMA becomes (K, 5, 7, 10, 27)

d. Process record against queries of awake users. It satisfies the queries of U1 and U3 so give the record to program A acting for U1 and program B acting for U3.

Repeat Step 5:

a. Low address in SIGMA is 5, which is in the same cluster, so we need not update awake/asleep lists.

b. Get the record at 5:

```
5
K2 6
K3 0
K4 0
```

c. Update SIGMA to (K, 5, 6, 7, 10, 27).

d. Give record to A acting for U1.

Repeat Step 5:

a. No action required.

b. Get record at 6:

```
6
K2 0
K4 0
```

c. The pointers of all keywords prime for
this cluster are null, thus no updating is needed for SIGMA.

d. Give the record to A acting for U1

Repeat Step 5:

a. Address 7 corresponds to a new cluster
   Awake becomes U1 and U2
   Asleep becomes U3

b. Get record at 7:

   7  
     K2 0
     K3 8
     K5 8

   c. All the pointers of prime keywords for this cluster are null, thus no updating is needed for SIGMA.

   d. Give the record to A acting for U1 and to A acting for U2.

Repeat Step 5:

a. Address 10 is in a new cluster.
   Awake becomes U1, U2, U3.
   Asleep becomes null.

b. Get record at 10:

   10  
     K2 0
     K5 11
c. All the pointers for keywords prime to this cluster are null.

SIGMA is (α, β, γ, λ, μ, ν, 27).

d. Give the record to A acting for U1 and to A acting for U2.

Repeat Step 5:

a. Address 27 is in a new cluster.

Awake becomes U1, U2, U3.

Asleep becomes null.

b. Get the record at 27:  27  K2  0

c. All the pointers for keywords prime to this cluster are null.

SIGMA is (α, β, γ, λ, μ, ν, 27).

d. Give the record to A acting for U1.

Repeat Step 5:

a. All addresses on SIGMA have been used, the algorithm is then terminated.

We can observe several things from this example. First, in selecting the permitted clusters from the
candidate clusters, the algorithm in Step 3 is in effect dynamically intersecting the user's query with the expression with a TYPE.5 specification which describes the permitted portion of the file. This minimizes the computational overhead by checking the protection of only those clusters which might include records which satisfy the query, and second by not expending any effort (i.e., prime keywords determination, etc.) on those clusters denied to the user. Second, the records which satisfy multiple users are directed to each pertinent user. No records need to be accessed more than once, since the algorithm processes all the lists in parallel. As we discussed earlier, keywords from X (here, the security keywords) are unlikely to be chosen as prime keywords. true for all of the records in a given cluster.

Additional Topics

We have attempted to avoid some of the philosophical aspects of the problem of protection. However, the work would be incomplete if we did not at least touch upon them. We shall consider two issues: what constitutes an access violation; and how alteration of the protection pattern may be controlled.
Access Violations

A point that is somewhat hidden in the engineering model is the determination of when an attempted access violation has occurred. This comes about because the system, in effect, expands each query and thus may request access to clusters denied to the user. The following example illustrates this problem.

Example: Access Violations

Consider a file with the following structure. S2, S3, and S4 are security keywords; K1 is not a security keyword but is a keyword for the file.

![File structure diagram](image)

Figure IV.15 - File structure.

Let the following protection specification be in effect:

\[
\text{TYPE.5((U1),S2, deny)}
\]

Thus, the user is denied access to clusters 1 and 3. Now,
consider a query of $K_l$. $K_l$ is true in all three clusters, but the user is permitted access only to cluster 2. Is this query an attempted violation? Certainly the user has (albeit unwittingly) attempted access to clusters 1 and 3 which are denied to him. Since there was a cluster to which he was permitted access that may contain records satisfying his query, one can argue that this was a legitimate query.

Actually the situation is more complex than this example can show. What we have are four distinct possibilities of what may occur when a user makes a query:

1) There are no clusters in the intersection of Step 1 in the parallel search algorithm, and thus no possible records which satisfy the query.

2) The clusters in the intersection are all permitted.

3) Some of the clusters in the intersection are denied and some are permitted.

4) All the clusters in the intersection are denied.

Clearly the first two cases are the result of legitimate queries, although in some systems we may wish to maintain an audit trail of even these queries. For a keyword that is present in a number of records, case three will result fairly frequently. Some systems [ConwR72a, ConwR72b,
HsiaD68a, HsiaD68b, ManoF711 consider case three as the normal mode. Different users making the same request will receive different responses because of their different access rights. However, in a highly suspicious system, case three might be construed to be an access violation attempt. Our natural inclination is to consider case four to be an access violation. Certainly this would be the result if the user had made a query which was, for example, identical to that of a TYPE.5 protection specification describing denied accesses for that user. However, even these cases are not always clear cut access violations. Consider a query of S3. Nothing has been said in the protection specification about such a query, but the only times that S3 occurs is in combination with S2, which is denied. At this instant S3 acts like a denied keyword. No query containing S3 will result in records being returned to the user, even though we have said nothing explicitly about S3.

In summary, because we allow for broad queries, the exact nature of access violations is unclear. Depending upon how suspicious we as system designers are, and how much time and effort we are willing to spend in response to these suspicions, we may wish to analyze requests more completely when the request results in cases three and four.
Most of the theoretical work in protection mechanisms considers the problem of protecting the protection information (the "(Protection)\#" problem) in great detail. This is because with the right to alter protection of an object the careless or unscrupulous user can circumvent the most carefully designed protection system. (A variant of GIGO: "garbage in, garbage out".) Tightly interwoven with the "(Protection)\#" problem is the problem of consistency, how can we maintain order in a system while allowing users to have considerable flexibility in the defining protection of the system's data objects.

How can such a system allow for the modification of the protection pattern in controlled ways? Most important perhaps is to limit the authority of users to subsets of the total protection pattern. Thus, if a mistake or malicious act occurs, the potential damages can be contained to a single compartment. This compartmentalization is also observed in the rest of the data base through the utilization of clusters. How then do we control the parcelling out of rights to alter subsets of the protection pattern? What guidelines should be followed, who should have access to what?

Consider two situations. An employee has just been granted a Top Secret clearance and it is desired to give him...
access to our Top Secret data so he can continue with his work. Case two is that a penetrator has determined how to alter protection and desires to give a cohort access to Top Secret data. In each case the actions at the system/user interface are identical. A (seemingly) legitimate request has been made to give more permitted accesses to a user who has not previously had such rights. With more information could the system distinguish between these two cases and refuse to allow only the penetrator's request? By forcing such a request to be "countersigned" by some authenticating officer, the system could achieve improvements in this area. However, this is a little more than postponing the issue, no fundamental improvement is actually made. If the authenticator blunders or is contravened, the results are the same as those occurring when no authentication is done. It is questionable whether the authenticator is any less likely to err than the originator. Still, it is probably a good idea to have such checks, if only as an attempt to force any conspiracy to include more members, risking its disclosure.

Ultimately any security system must rest upon the integrity and concern of individuals. If we give our trust to an individual unworthy of such trust, no system can hope to absolutely prevent the misuse of that trust. Any attempt
to "second guess" the users authorized to alter protection will be vulnerable in precisely those areas in which we desire the least vulnerability, enlarging a user's privileges. The best we, as system designers, can hope to do is to first record all such changes for subsequent review and second to force many different individuals to concur if a violation is to be consummated.

Context Sensitive Protection

So far we have limited our consideration to those protection specifications (and thus those protection patterns) which depended upon only the access being attempted, which we called context-free protection specifications. Let us now enlarge our considerations to include specifications which depend upon additional factors such as previous accesses. Because this topic represents several meaningful theses by itself [BaumR74, NeeCJ74, NeeCJ75, HartR75], we shall limit our discussion to relating how certain specific previous papers may be related to the current work.

**Defn:** A context-sensitive protection specification is one in which the decision to permit or deny an access attempt requires more information than the \((u, a, d)\) triple describing the attempted
access.

Obviously, to consider unrestricted context sensitive protection patterns would be hopeless, since we could not restrict the domain of the specification. We shall consider only those context-sensitive specifications which rely upon environmental factors, e.g., time of day, terminal being used, etc. and upon historical information about previous access attempts, both permitted and denied.

Suppose that a user has given us the following protection requirement: "If any user has read this file, he must not be able to copy it into a differently classified file." We cannot accommodate this type of rule with context-free specifications without being overly restrictive. What has to be done is to first identify the fact that the sensitive file has indeed been read, and to change the protection pattern to deny write access to differently classified files. This has been called the #property by researchers at the MITRE Corp. [BellD73a, BellD73b, Schl73] and has been independently considered in greater generality by Hsiao and Nee [NeeCJ74, NeeCJ75]. Two major areas must be addressed to incorporate this type of requirement into our model. First, how do we determine that an access to the sensitive file has been accomplished, and second, which accesses are now to become denied. The
detection of the access can be achieved by the use of the node procedures suggested in the data base model. Since we have already partitioned the records of the data base into groups which have homogeneous security, the detection of accesses to a group is relatively straight-forward. We first must examine the various access types in the system to determine which are information extracting and which are information storing. A simple "read" access is information extracting. A "write" access is information storing. An access type may be both information extracting and storing, as for example "update", where it is necessary to first read records of the file to determine which should be modified, and afterwards to perform the desired modification. So, the *-property becomes: "When an information extracting access is requested and permitted, all information storing accesses to "lower" classified data must become denied." One problem is to determine which data have "lower" classifications. If we have a hierarchical classification scheme, like the military scheme modelled by Bell and LaPadula [BellD73a, BellD73b], this determination is trivial. In our more general model, the determination is not so obvious. (Nee allows even more generality, no restrictions being placed upon the "triggering access" or the accesses to be changed [NeeCJ74].) The individual using such context-sensitive protection specifications must give an ordering (or at least
a partial ordering) relation for the various logical equivalence classes. Notice that to implement such context-sensitive specifications, it is necessary to allow for the dynamic alteration of the protection pattern. Above, we have pointed out the obvious vulnerability of the protection pattern. We are again getting into the "(Protection*)" problem, so we will not suggest general solutions. One stopgap measure would be to allow context-sensitive specifications to alter the protection pattern in such a way that the new pattern $P^*$ is a restriction of the existing pattern, $P$. (Recall that $P^* \leq P$ means that there are accesses denied by $P^*$ but permitted by $P$ and none which permitted by $P^*$ and denied by $P$.) A thus restricted protection specification could only deny accesses previously permitted. Another potential restriction designed to ease global service denial threats (i.e. one aberrant user affecting all other users) could be to restrict the effect of the specification to the user who triggered it. Thus, the untoward actions of one user would likely have small effect on the entire community.

Summary

We have presented a model for a Data Secure System at an intermediate level between the structural level and the
implementation level. The concepts of security atoms and clusters form the basis for the model. By organizing a database via clusters we are able to minimize the number of access decisions which need to be made, yet we do not compromise the security. Since the TYPE.5 protection specifications developed in the conceptual model are utilized directly, it should be obvious that any protection pattern can be faithfully carried out in the engineering model. We have also demonstrated the correctness of our algorithms in an informal way. In the next section we shall discuss the salient points of an actual implementation of these algorithms.
Chapter V. Correctness and Completeness of a Basic Data Secure System

Introduction

When one embarks on a research effort in data base management systems, he must eventually demonstrate the theoretical results in a practical way. It is therefore necessary to implement a demonstration system. Yet, to produce a full-size data base management system is a costly and time consuming effort [GerkB72] which is far beyond the scope of typical dissertation research.

To demonstrate the concepts of the preceding chapters a small system was constructed. This system, which we shall refer to as the Basic Data Secure System, BDSS, is intended to serve three major purposes. First, it gives a concrete presentation of the algorithms developed in the engineering model. Second, it allows for some limited initial experimentation with various user interfaces, particularly in the area of protection specification definition. We have found, for example, that the creation of a consistent set of protection specifications is really a rather formidable
task, even using the TYPE.5 specifications of the structural model. This result lends additional impetus to the work of Hartson in languages for specifying protection [HartH75]. Finally BDSS serves as a model for a technique of program analysis and certification which appears to hold more practical potential than traditional program proving. This technique, which we shall call failure mode analysis, allows the designer to focus on a limited number of crucial factors in the system. When these factors are considered, the system is shown to possess specific properties, in our case, that no denied records are released. Although failure mode analysis is less formal than program proof techniques it has its roots in the same concepts. We feel that the failure mode technique can offer considerable assurance of the fact that a program meets its design specifications without requiring the extremely precise and complex analysis associated with program proving [ElspR72, DeutP73, Lave74, HorgW75].

No system is ever implemented in an ideal environment. BDSS is certainly no exception. Like many other projects our major limitations were manpower, machine capability and language flexibility. Initially BDSS was conceived as a demonstration of the algorithms from the engineering model.
There would be another, larger and more complete system with which actual experiments would be performed [KaffN75] [*]. During the course of this effort, the scope of BDSS increased to be much more of an actual system than was originally intended. Limitations in the design and implementation which were acceptable when the system’s role was smaller became quite cumbersome when its scope expanded.

One of the most important of these decisions was the

[*] As mentioned in the introduction there are six major tasks:
1) Data Secure Systems and deadlock based security mechanisms. [This document, KaffN74]
2) Context protection and consistent control in data secure systems. [HeeCJ74, NeeCJ75]
3) Data secure computer (hardware) architectures [BaumR74].
4) Design and certification of data secure system. [Horgi75]
5) A system for experimenting with access control mechanisms. [Kaffi75].
6) Languages and language translators for protection specifications. [Hartii75]
choice of programming language. We decided from the very beginning to use structured design techniques throughout. Thus, the algorithms of the engineering model are really the highest level descriptions of the system. The target system, a Digital Equipment Corporation, DECsystem-10, had several different languages suitable for a structured design. However, the particular choice of language, ALGOL, turned out to be somewhat poor. The most important reason is that the ALGOL run-time system is exceptionally primitive, lacking even the power of FORTRAN. It is somewhat unfortunate that such a mismatch of language power to run-time system power has occurred, since many of the features of DECsystem-10 ALGOL are very well-suited to system implementation. The most significant limitation of the ALGOL run-time system was its lack of support for direct access storage devices (DASD). These devices had to be handled in a strictly sequential manner, obviating much of the effectiveness of using DASD. Another significant problem area was the lack of a simple way to trap I/O errors. Typing in a bad character was a fatal condition, a totally unacceptable situation in a user-oriented system. Finally, it was a nuisance to do even simple I/O in ALGOL.

With these limitations in mind, let us get on with some discussion of the system as built. We shall dwell on two
main points, system completeness and correctness.

The Completeness of the Basic Data Secure System

To begin our discussion of system completeness, we should define more formally what we mean by the term.

**Definition:** A Data Secure System is **complete** if every record can be potentially protected differently for each user and access type.

We use this definition to emphasize the potential flexibility that a Data Secure System must possess. If a system is complete, then any less elaborate protection requirement can obviously be met as a special case.

The system achieves a high degree of completeness. To demonstrate this assertion we must show two things. First, that all legitimate protection specifications may be input to the system. Second, that when records are inserted into the data base, each of the clusters continue to represent only one logical equivalence class. This second requirement is also necessary to avoid data base reorganization if the protection pattern is changed.

In BDSS these two factors are satisfied by the actions of two modules, INSERT and SPECS. (Listings and other program documentation comprise Appendix A.) INSERT does the
actual record insertions, and SPECS processes the cluster descriptions (i.e., descriptions of the logical content of the cluster) against the existing protection specifications to assign protection to the cluster. These descriptions of the logical content are the keyword expressions which determine the logical equivalence class. Recall that we stipulated the existence of a set of keywords, $X$, upon which the logical equivalence relation, $\text{LOG}$, would be based. Each of the canonical form expressions of keywords from $X$ describes a potential $\text{LOG}$ class. Under the definition of cluster equivalence, $\text{CLS}$, in the structural model, all the member records of a cluster belong to the same $\text{LOG}$ class. Of course, because of the other components of the $\text{CLS}$, cluster equivalence, relation, there might be several clusters associated with the same $\text{LOG}$ class.

SPECS uses an almost direct mechanization of the TYPE.5 protection specifications. This type of protection specification has the form

\[ \text{TYPE.5}((\text{users}); Q, \{\text{deny/permit}\}) \]

where $Q$ is a Boolean expression of security keywords.

The operation of SPECS is the most straight-forward one. After initializing its internal descriptions of the protection specifications, the program loops through each of the cluster descriptions and processes each one against the
entire set of protection specifications. If the cluster
description satisfies the "query" of a protection
specification, the appropriate portion of the protection
pattern is filled in. This is done by setting values of an
access matrix. A check is also made for consistency. If
two specifications are inconsistent, all the contested
accesses are denied, and a message is printed describing
where the questionable situation has occurred. When all the
cluster descriptions have been processed, the access matrix
is written out into a file. This file is later used as
input to the data base accessing parts of the system.

SPECS is complete in the sense that no restrictions are
made upon the protection specifications that are input,
other than that Q is restricted to be a single conjunct.
Because all the specifications that comprise the protection
pattern are, in effect, OR'ed together, this single conjunct
restriction does not impose any limitations on the
completeness of the system. Any arbitrary Boolean
expression of keywords may be expanded into a disjunctive
form. SPECS accepts the conjuncts of such a DF with no
restrictions. Thus, we assert the obvious result that SPECS
is complete under our definition of the term. That is, any
arbitrary protection specification may be input and no
cluster need be broken up to afford it the protection
desired. This does, however, assume that $X$, the basis for LOG, includes all the keywords which may be used in protection specifications. This requirement was initially stated in the engineering model, and does not really represent a severe constraint.

The major impact of INSERT on completeness is that it must not cause the contamination of a cluster by the insertion into it of records belonging to a different logical equivalence class. This also, of course, has an impact on the system correctness, which we shall consider in more detail later. The cluster description of a candidate cluster is compared with a similar description derived from the record to be inserted. The insertion in a given cluster is allowed to occur only if these two descriptions are logically equivalent. If no such cluster can be found, then a new cluster is established. The cluster description for such a new cluster is established simply by copying the description from the first inserted record.

So, since INSERT preserves the integrity of the clusters, and since SPECS allows any consistent set of specifications to be used, the BDSS is complete.
The Correctness of the Basic Data Secure System

In discussing the verification of a system it is sufficient to give a formal proof of its correctness. Although systems of similar magnitude have been formally proven correct [GoodD74, SchL73] by special efforts, there is no systematic and economical way to develop proofs for systems of the size of typical data base management or operating systems. We shall, therefore, not resort to the formal proof technique. (In a parallel task Horger [HorW75] is developing formal proofs of programs similar to but significantly less complex than BDSS.)

On the other hand, it is unreasonable to expect that the reader is willing to accept the correctness of BDSS without some strong motivation. What we shall do is to introduce a methodology which is at neither the extremes of the formal proof technique or of totally informal arguments.

The method, which we shall call failure mode analysis, has some distinct advantages over other avenues of approach. The verification of the system is quite specific in that simple conditions need be demonstrated to assert that the system behaves correctly.

Our approach to correctness will be similar to other efforts [WeihD74, WeisC74] in that we shall make hypotheses
about system protection failure modes and then demonstrate how each mode is precluded. (Notice that in many ways this approach relies upon the ingenuity of person performing the analysis, just as considerable ingenuity is needed to develop a correct proof.) What we hope is that by being systematic in our approach, we can convince the reader that the protection features of BDSS are correct. In this work we are only concerned with failures of the database management system protection mechanisms.
Denied Record
Released

Access attempted in
cluster denied to
all active users

Error in
inserting

Error in
searching

Awake/asleep
failure

Internal table
garbling

Error in
inserting

Error in
searching

Record LOG class
Cluster LOG class

Incorrect
Keywords
Selected
for
Classification

SATISFY CLUSTERS
Failure
Failure

SPECs
Failure

INIT.PROT
Failure

GIGO - SPECs
or INIT.PROT
Failure

CLUSTERS
Failure

Disjoint
List
Failure

Internal
Cluster
Property
Table
Garbling

GIGO

Figure V.1 - BDSS protection failure modes.
In Figure V.1 we show the BDSS failure mode tree. As a final provision before we begin discussion of the tree, this analysis assumes that the input of records to be inserted and of protection specifications to be processed into a protection pattern is "correct". That is, that the records are an accurate representation of reality and that the specifications truly represent the intent of their initiator. We shall revisit these two areas in the summary, where some possible research topics will be suggested. In the following discussions some familiarity with DECSYSTEM-10 ALGOL is assumed.

Disjoint List Property

Since the disjoint list property is a crucial underlying point, we shall begin there. What must be shown is that every list has all its member records in one and only one cluster. In BDSS this is a relatively trivial exercise. The K-pointers are actually relative addresses within each cluster. Thus, no real addresses are used. This is a direct result of basic design decisions in the DEC TOPS-10 operating system (DECSYS1). Its designers chose to insulate the user almost totally from the physical characteristics of the system, presenting a standard logical
The interface for all devices. The standard DECsystem-10 disks are a block oriented device, and the disk service routines mask from the user where the blocks of his file are actually stored. Thus, it is unusual (and somewhat difficult) to utilize the physical block addresses. In ALGOL, even the block structure of the files is hidden from the user by the run-time system. So, since it is hard to use actual addresses, relative addresses are used. Since obviously, a relative address cannot point outside a cluster, the disjoint list property is maintained. In BDSS, the compartmentalization of the cluster concept is utilized by making each cluster a distinct file.

The Correctness of SATISFY

The module SATISFY is used in a great many places. Its function is to determine whether a record (or more specifically the keywords of a record) satisfy an arbitrary Boolean conjunction of keywords. Specifically, SATISFY is a Boolean procedure with the following arguments:

```
SATISFY ( FILE, N.FP, TERM.KWD, NO.OF.KWDS, I, J)
```

- **FILE** - a string array, each element is a keyword
- **N.FP** - number of keywords in FILE (i.e.,
index runs from 1 to N.FP)

TERM.KWD - a three dimensional string array
which is a complete query.
Each element is a keyword.
The J-th term from user I is found as
the keywords TERM.KWD [I,J,I]
through TERM.KWD [I, J, NO.OF.KWDS].

SATISFY returns true if all of the positive keywords and
none of the negative keywords from the J-th term of the I-th
user are found in FILE. This is done by incrementing a
counter, M, on each successful search of FILE for an element
from TERM.KWD. (In the case of negated keywords the search
is successful if the keyword is not found.) The actual
search is done by another module SA.FIND:

SA.FIND(N.FP, FILE, TST)

SA.FIND returns true if and only if there is an 1 ≤ L ≤ N.FP
such that FILE [L] = TST. M is initialized at 0 and is only
incremented for successful searches. Thus, the only way M
can be greater than or equal to NO.OF.KWDS is for the search
for each of the keywords of the conjunct (the J-th from user
I) to have been successful. SATISFY returns true, then,
only when FILE does indeed satisfy the conjunct.
The Correctness of CLUSTERS

Another module which is used in a variety of places is CLUSTERS. This module has the function of returning a list of candidate clusters, THETA, for a given conjunct of the query. It does this by intersecting together the cluster identifiers for each of the non-negated keywords of the conjunct. CLUSTERS returns only candidate clusters to which the user is permitted access. In our analysis of how this function was to be performed, it was determined that a slightly more sophisticated algorithm should be used. First, an initial pass is taken through all the non-negated keywords of the term to determine the keyword found in the smallest number of clusters. These clusters are used as the initial THETA. (We will continue to use the terminology of the algorithms from the engineering model, as was done in the programs.) From this THETA, we delete (by setting THETA[I]:= -1) any cluster to which the user is denied access. Then, the intersection of the sets of cluster identifiers for all the other non-negated keywords of the term is performed. This technique starts with a small (i.e., few member) THETA as the initial case. The computational work necessary to do an intersection of two unordered lists of lengths m and n is in the worst cases proportional to
169

By the initial work we do to start with a small THETA, \( n \) is minimized.

The most important aspect of the correctness of CLUSTERS is that it never returns as a member of THETA a cluster to which the user is denied access. (Note that in the single user case this is a sufficient condition for the prevention of the release of denied records.) We are assured of this condition because of two facts: First, no clusters are added to THETA after its initialization. Second, immediately after the initialization all the denied clusters are deleted in a simple pass through THETA.

Correct Insertion of Records

Generation of Correct Record Descriptions.

With the correctness of SATISFY and CLUSTERS demonstrated, we can now discuss the correctness of the record insertion process. Records to be inserted are gotten from the user by the module GET.INSERT. A record would be mis-classified if its canonical description (i.e., the keywords of the record that are also members of \( X \)) were incorrectly specified. In GET.INSERT each keyword of the
record is checked against the keywords of \( X \). If the record keyword is found in \( X \) then that keyword is moved into \( \text{TERM.KWD} \) as a keyword from user 1, term 1 (\( \text{TERM.KWD}[1,1,x] \)). Thus, at the exit from \( \text{GET.INSERT} \) each of the keywords of the record that are also in \( X \) are formed into a conjunct identical to that of a query. (If there are no such keywords, the artificial keyword "\( \text{UHCLASSIFIED} \)" is inserted in the record and \( \text{TERM.KWD}[1,1,1] \).) The counter \( \text{NO.OF.KWDS} [1,1] \) is incremented as each keyword is inserted into \( \text{TERM.KWD} \). So at the return from \( \text{GET.INSERT} \) we have all the basis keywords in \( \text{TERM.KWD} \) and a count in \( \text{NO.OF.KWDS} \).

**Actual Record Insertions**

The next step is a call to the module \( \text{INSERT} \) which actually does the record insertion. In \( \text{INSERT} \) the first step is to call \( \text{CLUSTERS} \) with the access control array replaced by a similar array indicating which clusters are full. This is in keeping with the discussion in the engineering model about the similarity of access types. We stated, "An access type of insert should be denied to target clusters which are full." By replacing the array \( \text{DENY} \) with \( \text{FULL} \) in a call to \( \text{CLUSTERS} \) this is precisely what is done. (Incidentally, this opens a philosophical problem which we shall discuss briefly later.) As we have shown, \( \text{CLUSTERS} \)
returns an array THETA containing the identifiers of permitted (in this case not full) candidate clusters. Since the keywords of the query here are all from X, and within a cluster every record belongs to the same logical equivalence class, we know that the records of the clusters in THETA at this point all have the same basis keywords true as the inserted record. However, some members of THETA may have more basis keywords true than the inserted record. For example, consider the following case.

Cluster 1: K1 & K2 & K3
Cluster 2: K1 & K3 & K5
Cluster 3: K1 & K3

Basis keywords of inserted record: (K1, K3)

All of these clusters would be returned from CLUSTERS, but only cluster three would be suitable for insertion under the requirement that a cluster have only one LOG class represented.

This problem could be solved many different ways. One could simply fetch a record from each of the clusters in THETA and compare the basis keywords between it and the inserted records. Another choice would be to include negated keyword information in the directory. We have chosen a third method. An explicit characterization of the logical content of each cluster is maintained. For each
cluster in the system the basis keywords which describe it are found in an array CLSTR.EXP. This array is structured identically to TERM.KWD so SATISFY may be used to see if the inserted record satisfies the canonical cluster expression. If not, then that cluster is not suitable for insertion. The combination of CLUSTERS and SATISFY mean that THETA will contain only the identifiers of clusters which meet all the criteria for insertion: 1) room for the insertion and 2) the same LOG class. Furthermore, THETA will have all such clusters.

If, at this point, no clusters are left in THETA, a new cluster must be established. Due to the characteristics of the TOPS-10 system mentioned earlier, it is quite difficult to get at the physical realities of the system. Thus, the concept of cell equivalence is not employed in BDSS. No attempt can be made to place clusters representing the same logical equivalence class into the same cell. What was done was to let each cluster be a separate file. So, when a new cluster is started, the user must supply a new file name. The CLSTR.EXP entries for this new cluster are copied directly from TERM.KWD, the canonical description of the inserted record. CLSTR.EXP is therefore consistent, because nowhere else are entries inserted.
INSERT then makes up the pointers for all the inserted records keywords, adjusts the directory, and writes the resulting record out. None of these affect security because we have assured that the record is being inserted into the correct cluster. Of course, programming errors here may have bad effects upon the searching file.

So, because INSERT uses CLUSTERS and SATISFY, and these modules have been demonstrated correct, we assert that INSERT is also correct.

Correct Searching

The Parallel Search Algorithm

The security of the parallel search algorithm, PARA, rests on two points. The first is that CLUSTERS is used to generate the list of candidate clusters. For a single user system this would be sufficient because searching would never be initiated in denied clusters. To emphasize the parallelism of processing in different clusters, Step 5 of the parallel search algorithm is actually done in a loop on a master list of clusters. This master list is formed by merging together the cluster identifiers returned from CLUSTERS for each of the terms in the entire multi-user query. Thus, MASTER.THEETA contains the identifier of each
cluster in which searching will occur. So our original algorithm would then become

\[
\text{FOR } \ i = 1 \ \text{UNTIL L.MASTER.THETA DO}
\]

Step 5 for cluster = MASTER.THETA[I];

This also means SIGMA is initialized only for a single cluster and never grows to be longer than the number of records in a cluster. By an argument similar to that of the previous chapter, in the single user case, no addresses for denied records would ever appear on SIGMA. Also, then, MASTER.THETA never contains any denied clusters.

In the multi-user case, the second aspect of protection for the parallel search algorithm is used. This is the awake/asleep mechanism discussed in the engineering model. A record is examined to see whether it satisfies the user's query only for those users who are permitted to access records in this cluster. For the others, the record is not even processed. Because only one cluster at a time is processed, no records will be released to users that are denied access to them. In the single user case this could be used as a redundant check. In BDSS, this redundancy is actually not present since both PERMIT, the array used in the awake/asleep mechanism and DENY, which is used by CLUSTERS, are initialized from the same source.
The final protection failure mode which we shall discuss is GIGO, "garbage in--garbage out". Under this we shall include those modes which tend to garble the protection decision information between the time of its initial utterance in the creation of the protection specifications until its eventual use.

The first of these GIGO modes occurs in SPECS. The possibility exists that the access matrix will not be filled up correctly due to either an error in reading the cluster canonical expressions or in processing them against the protection specifications. The canonical expressions of keywords that describe a cluster are read from a file, CELLS.NEW, that is produced by the updating and searching system. These expressions are generated when a new cluster is started and are never modified elsewhere. They are simply read in one at a time into a string array CE, which is then used identically to FILE in a call of SATISFY. The protection specifications are maintained in a form almost identical to the way in which they were input. Each cluster expression is processed against all the existing specifications in a series of calls to the previously demonstrated module SATISFY. Thus, we assert that SPECS fills its internal access matrix correctly. The resultant access matrix is written into a file, PROT.DAT which serves
as input to the update and search system.

Another GIGO failure mode occurs with the reading of PROT.DAT as the update and search system is initialized. The module INIT.PROT essentially copies the information from PROT.DAT for the active users of the system. PERMIT and DENY are initialized such that if there is an active user who is not among those mentioned in PROT.DAT, that user will not be permitted any accesses. The other GIGO failures relate to the cluster expressions and to the directory. The directory save and restore procedures are almost identical, so this type of failure seems remote. In order for the system to function at all, the cluster expressions and the actual file names corresponding to each cluster must be read in correctly, so that a failure here would likely result in an overall system failure.

This exhausts the failure mode tree shown in Figure V.1. Because these arguments have been informal, it cannot be stated absolutely that BDSS is formally correct. However, the analysis has been thorough enough to warrant considerable confidence in the assertion that indeed there is no way in which BDSS can return a record to a user when that user has been denied access to the record by a protection specification.
The failure mode analysis technique can be viewed as being similar to a proof sketch. Thus, it should not be considered as merely an ad hoc technique. If the extra time and effort is justified, the failure mode analysis can be expanded into a complete proof of correctness.

It should be remarked that only by the use of the structured design techniques used in BDSS, would such an analysis be possible. Because of the enforced modularity imposed by "GOTO-free" programming, the failure mode analysis can be succinct and thorough.

**Summary**

In this chapter we have presented a detailed analysis of the correctness and completeness of an actual system. Although the intended role of the system is small, it does serve to illustrate the algorithms of the engineering model in a concrete way. The failure mode analysis technique appears to offer some promise for system quality assurance in even large systems.
Chapter VI. Results and Future Research Topics.

Introduction

In the preceding five chapters we have taken the topic of highly secure database management systems from its basic conceptual foundations through the implementation of a demonstration system. It is the purpose of this chapter to attempt a summarization of the most significant aspects of the previous chapters and to suggest topics opened by this research for which further work would be advisable.

A Summary of the Results

Although the conceptual model was really designed to be a definitional framework, there are still some important points made there. For example, other researchers [GrahG72, JoneA73] have focused mainly upon the mechanization of access control. Their models, of necessity, were at a very low level and included much detail. In the conceptual model we introduced the notion of protection specifications. These are manifestations of access control policies rather
than their mechanizations. The distinction, though subtle, is important. It seems quite unreasonable to require the user to make statements about access control at a level of detail equivalent to our TYPE.1 protection specifications, as would be the case in some other research models of protection. The TYPE.5 protection specifications, although certainly not the ultimate in flexibility, are posed completely in the users view of the data base. We have argued that any protection specification can be expressed as a collection of TYPE.1 specifications, thus our user level TYPE.5 specifications can be expressed as TYPE.1 specifications. Our results can obviously fit into these lower level models. The user can make his statements about protection requirements at a high level, and the system can process these to yield the detailed low level information that is more suitable for use internally. We also introduced some primitive operators for the resolution of conflicting specifications. These operators allowed us to formalize the notion of deferral of conflicting access decisions.

In the structural model the basic concepts of the data base and system models were introduced. By choosing a general data base model, we are able to apply these results equally well to a variety of file structures that are
achievable as special cases of the general model. If a particular situation demands the use of, say, an indexed sequential file structure, all of our results are still applicable since such structures can be naturally achieved by the data base model. The data base model is also quite close to the relational model discussed by Codd [CoddE70, DatoC75] because our definition of a record is very similar to that of an instance of a relation (tuple). Thus, much of the work which has been done on these higher level models can be applied to our work. In particular, some of the difficult questions of consistency within records have been very well handled by relational techniques [HageS74, HeatI71]. One of the most important original results of this work is the concept of clusters. In previous work on this data base model [HsiaD74], only the cell equivalence relation was presented. With the new contributions of this work we have identified three separate file partitioning criteria. We have also given an explicit description of the notion of logical equivalence. Our results incorporate the work of Wong and Chiang [WongE71] and of Rothnie and Lozano [RothJ74] as special cases. We have also identified some parameters of the logical file design space in a way similar to that of Severance’s [SeveD70] work in the more physical aspects of file design. The file designer can systematically explore a portion of the file design space by
varying the cell size, the structural equivalence criteria, and the set of keywords upon which logical equivalence will be based; X. When these concepts are utilized, one would expect to see a file structure which has high retrieval precision and which tends to minimize the number of seeks required in accessing it. Finally, this work represents the first explicit recognition of the importance of the disjoint list property. While previous discussions have utilized this concept implicitly, we are the first, to our knowledge, to give an explicit characterization of this property and its importance.

The system model is based upon the concept of multi-user deadlocks. While this concept does not represent a viable alternative for direct implementation, it is shown to characterize a wide class of protection enforcement mechanisms. The ideas can be used to assess the completeness and correctness of possible protection mechanism implementations.

Hsiao [HsiaD75] has correctly pointed out that one must consider both the data structure and its associated accessing algorithms when discussing file models. In the engineering model we presented algorithms for the maintenance and accessing of the data structure developed as part of the structural model. Detailed examples of the use
of these algorithms were also presented.

One question which has come up several times during the course of this work is the number of clusters one would expect to see in a realistic data base system. We have presented some motivation that there are substantially fewer than $2^{2n}$ (where $n$ is the number of keywords in $X$, the basis of the logical equivalence relation). The question is really irrelevant for the following reasons. We have assumed that no controls may be placed on the protection specifications currently in force. We have also stipulated that due to compartmentalization requirements, all the member records of a cluster must have identically the same protection. It is this latter fact which allows us to make access control decisions for clusters. Finally, we stated that it is desirable that the data base need not be re-organized if the protection pattern is changed. The coarsest partition of the file which would in general satisfy these requirements is the partition induced by LOG, logical equivalence, with $X$ containing all the keywords which may be used in protection specifications, the security keywords. If any of the LOG equivalence classes were coalesced, then there exists a protection specification which would require the coalesced entity to be broken down back into its components in order to satisfy the uniform
protection requirement. Of course, if we can a priori exclude certain protection patterns (and thus certain specifications), it may be possible to utilize a coarser partition. One is obviously faced with a set covering problem in finding the coarsest partition, so attempting to do this would seem to be a questionable venture. So, in the absence of restrictions on the protection pattern, it does not matter how many clusters one will have, because attempting to cut down on this number may result in the violation of the basic system requirements.

We concluded by presenting the results of a limited experiment in the design and implementation of a Data Secure System. We also introduced an analysis technique, failure mode analysis, which can potentially serve as a significant tool in designing systems correctly. Under this technique a small set of critical factors is identified and a systematic analysis of the programs is performed to assure that these critical conditions are met. In a Data Secure System, the single critical condition was that no denied records be released to a user. We demonstrated through essentially proof sketches that this condition was met in the system.
**Future Topics**

One measure of the success of a particular piece of research is the number of new topics which are opened by the work. By this standard, our work has been quite successful. Let us go through the chapters and discuss some of the potential topics raised in each.

The conceptual model utilized a binary (permit/deny) algebra. Baum [BaumR74] has introduced a ternary algebra, the third value being "undecided". It would be quite interesting to explore this topic in some more generality, introducing a multi-valued logic. The major issue here would be the practicality and utility of such a formulation.

Another important issue is languages for the specification of protection. Hartson [HartR75] is making substantial progress here, and other researchers are reported to be working on this topic. In the minimal experiments performed with BOSS, the need for specifications of more power than even our TYPE.5 protection specifications became apparent. We also found that a modified definition for consistency might be needed. A user making a specification using only a local knowledge of part of the data base might easily create a specification that is not consistent, the conflict occurring with other more global
specifications. If the languages used were more flexible, this problem might be minimized.

Although this is a dissertation about protection, the structural and engineering models contain considerable potential work in the data base management area. The most general of these topics would be to create an explicit characterization of the design space for the data base. By identifying three of the axes (cell size, logical equivalence and structural equivalence) we have started in this area. Severance [SeveD70] has identified similar parameters for a somewhat lower level model. Once this multi-dimensional space has been characterized one could deduce a feasible region based upon the computer resources to be dedicated to the system, (e.g., amount of CPU power, size of main memory, amount of secondary storage etc.) and the magnitude of the application (e.g., size of the data base, number of terminals, response time, required ease of updating, etc.). Finally given a cost function one could systematically explore this design space to find an optimal (or at least an acceptable) solution to a specific data base design problem.

One area which we have only mentioned in passing is that of consistency. Here we mean consistency of the record to some objective reality. This legitimately falls into the
protection and security area when one considers the problem of preventing garbage in the database. Indeed, this would represent a very subtle form of attack of a data base system. One could attempt to prevent the extraction of legitimate information by obscuring it with fictitious data. (This, incidentally, is done deliberately with U.S. Census data in an attempt to make it difficult to obtain information about small groups of individuals [Hanski71].) Considerable work relating mainly to the artificial intelligence area could be done here.

We have considered protection of entire records based upon the content of specific fields (keywords). Some significant theoretical work is needed in the area of sanitization and declassification of records by the systematic deletion or alteration of specific fields. This is closely related to the record consistency problem, in that it must not be possible to infer the missing information from that which remains. Both topics deal with the relation between keywords within a record.

We concluded the engineering model with two open topics, context-sensitive protection and access violations. Since these were explored there, we shall not repeat the discussion here, other than to re-emphasize the overriding importance of these two areas.
Another topic which was touched upon in the engineering model was the problem of whether denial of access gives information. This is a question that opens a great many potential research topics. The major question relates to determining the user's state of knowledge. He may know so much that even to deny access gives some added information. Although, in general, this problem is obviously unsolvable, some research is needed to explore the limits of certain restricted cases.

Chapter V was a discussion of a small experiment in building a Data Secure System. Until larger and more complete systems are constructed, there will be legitimate questions about the validity of the experiment.
Appendix A. BDSS Documentation.

Because of the nature of the BDSS implementation, the programs are largely self-documenting. Thus, we include the listings of the programs themselves rather than the more traditional flowcharts. In Figure A.1 the structure of the system is shown. Following this figure are the program listings.
Figure A.1 - BDSS Module Structure
BEGIN CHECK 11;

STATE ARRAY DIR.OPS[1:54], FILE[1:10], CLUSTER(1:4), SEC.KDS[1:24], VP[1:5], CLSTR.EXP[1:11, 1:2, 1:2];

LARGER ARRAY DIR.OPS[1:44], DIR.PNS[1:54, 1:61], FILE.PNS[1:101];

TOUR ARRAY FILE[1:11, 1:2];

LARGER II FILE.PNS: AVP, NO.OF.SEC.KDS, NO.OF.DIR.KDS, NO.OF.CLUSTERS, REC.P.CLUSTER, NO.OF.USERS, I, J;

STABLE AVG;

EXTERNAL PROCEEDURE DIR.Put;

EXTERNAL PROCEEDURE PUT;

EXTERNAL PROCEEDURE EMPTY;

EXTERNAL PROCEEDURE MARK;

EXTERNAL PROCEEDURE GET.LIST;

EXTERNAL PROCEEDURE INIT.PLS;

EXTERNAL PROCEEDURE FILL;

EXTERNAL PROCEUREMENT DIRECTORIES;

EXTERNAL CLUSTER 3;

INPUT("TRY")\ INPUT("OS")\ INPUT("DS")\ INPUT("DS")\ INPUT("OS") OF DIR.KDS;

FILL(SEC.KDS, NO.OF.SEC.KDS, "SEC.KDS") OF FILE.PNS;

FILL(CLSTR.EXP, NO.OF.CLUSTERS);

SELECT Thứ 4 ; SELECT INPUT(4) ;

BREAK OUTPUT;

BEGIN

FILE.KDS ARRAY NO.OF.TERMS[1:10.OF.USERS];

BEGIN CHECK 11;

BOOLEAN ARRAY PERMIT[1:10.OF.USERS, 1:2]; DENY[1:10.OF.USERS, 1:20];
LEV1.ADJ  12-FEB-75  11:57  PAGE 2

3111)  STRING  ARRAY  USER. ID[1:10],OF USERS],TERM.KD[1:10],OF USERS], 1:5,1:6];
3211)  FOR T1 = 1:UNTIL NO. OF USERS DO
3311)  :   3351)  WRITE("ID OF USER =") ; PRINT(T,1,2);:
3351)  WRITE("-");  BREAKOUTPUT:
3361)  PRINT(USER.ID(T));:
3411)  END:
3521)  INIT.PROC(USER.ID,NO. OF USERS,PERMIT,DEAY);:
3541)  AGAIN:=false:
3411)  WHILE AGAIN DO
3411) BEGIN
3421)  SELECTOUTPUT(1); SELECTPUT(1):
3431)  WRITE("(C)QUERY,I:ERT,EXIT (0/1/5)-") ; BREAKOUTPUT:
3441)  RENAMES:
3451)  IF AIS="1" THEN
3461)  EXIT:
3471)  GET, INSERT( TEN.KD, FILE, AVP, D, FILE, PT, SS, R, AVP, NO. OF KIDS, NO. OF SEC.KIDS, SEC.KIDS):
3481)  INSERT(FULL.FILE, FILE, PT, SS, FILE, PT, SS, CLSTR, EXP, 1, CE, NO. OF CLUSTERS, CLUSTER, TEN.KD, NO. OF KIDS, NO. OF DIR.KIDS, AVP, 1, AVP):
3491)  END:
3511)  IF AIS="0" THEN
3521)  EXIT:
3551)  QUERY(USER.ID, NO. OF USERS, NO. OF TERMS, NO. OF KIDS, TEN.KD):
3531)  PARA (NO. OF USERS, NO. OF TERMS, NO. OF KIDS, TEN.KD, NO. OF CLUSTERS, REC.P.CLUSTER, DIR.KIDS, Dir.
3571)  PERMIT, DEAY, USER.ID, NO. OF DIR.KIDS, CLUSTER):
3541)  END:
3551)  IF AIS="11" THEN AGAIN=FALSE:
3561)  EXIT:
3571)  DIR.PUT(DIR.KIDS, DIR.PTS, NO. OF DIR.KIDS):
3581)  EMPTY(CLUSTER, FULL, 1, CE, CLSTR, EXP, NO. OF CLUSTERS):
3591)  END:
3611)  SELECTOUTPUT(1); WRITE("BYE! (C)"):
3611)  CHECKOFF 1:
3621)  END:
PROCEDURE QUERY (USER.ID, NO. OF USERS, NO. OF TERMS, NO. OF KWD'S, TERM.KND) ;

INTEGER NO. OF USERS;

INTEGER ARRAY NO. OF TERMS, NO. OF KWD'S;

STRING ARRAY USER.ID, TERM.KND;

! ENTER QUERY;

BEGIN;

CHECKON 1;

INTEGER I, J, K;

BOOLEAN ALLTRUE;

EXTERNAL BOOLEAN PROCEDURE NEGKND;

SELECTINPUT(J); SELECTOUTPUT(J);

WRITE("ALL KEYWORDS MUST BE QUOTED. NEGATED KEYWORDS START WITH -. [C]");

FOR I = 1 UNTIL NO. OF USERS DO

BEGIN;

WRITE("ENTER NUMBER OF TERMS IN YOUR QUERY "); WRITE(USER.ID[I]); WRITE("-"); BREAKOUTPUT;

READ(NO. OF TERMS[I]);

I LOOP FOR EACH TERM FOR USER I;

FOR J = 1 UNTIL NO. OF TERMS[I] DO

BEGIN;

WRITE("ENTER NUMBER OF KEYWORDS IN TERM"); PRINT(J, 3); SPACE;

WRITE(USER.ID[I]); WRITE("-"); BREAKOUTPUT;

READ(NO. OF KWD'S[I, J]);

ALLTRUE = TRUE;

FOR K = 1 UNTIL NO. OF KWD'S[I, J] DO

BEGIN;

WRITE("ENTER NEXT KEYWORD IN TERM"); PRINT(J, 3); SPACE;

WRITE(USER.ID[I]); WRITE("-"); BREAKOUTPUT;

READ(TERM.KND[I, J, K]);

IF NOT(NEGKND(TERM.KND[I, J, K])) THEN ALLTRUE = FALSE;

END;

END.
PROcedures GET, INSERT, DELETE, ARP-AP, H-AP, SEC-KIDS, OF-SEC-KIDS

BEGIN

PROCEDURE GET: IN: FILE-AP, H-AP, OF-SEC-KIDS
BEGIN

PROCEDURE INSERT: IN: FILE-AP, H-AP, OF-SEC-KIDS
BEGIN

PROCEDURE DELETE: IN: FILE-AP, H-AP, OF-SEC-KIDS
BEGIN

PROCEDURE ARP-AP: IN: FILE-AP, H-AP, OF-SEC-KIDS
BEGIN

BEGIN

PROCEDURE SEC-KIDS: IN: FILE-AP, H-AP, OF-SEC-KIDS
BEGIN

END

END

END

END

END

END
LEVIA.ALG  12-FEB-75  12:47  PAGE 4

1. FILE::PT::S::FILE::PT::S+1;
2. IF x::S[1,1] 1 THEN
3. FILE::I::FILE::PT::S+1::FILE::PT::S+1::"UNCLASSIFIED";
4. END;
5. WRITE("ENTER NUMBER OF NON-EXACT (A,V) PAIRS--")::BREAKOUTPUT;
6. READ(A,AVP);
7. FOR L = 1 UNTIL U = AVP DO
8. BEGIN
9. WRITE("NEXT (A,V) PAIR--")::BREAKOUTPUT; READ(AVPL);
10. END;
11. CHECKOFF 1;
12. END;
PROCEDURE PARA (NO. OF USERS, NO. OF TERMS, NO. OF KNOs, TERM.KNO, H.CLSTRS, REC.P.CLUSTER, DIR.KNOs, DIR.PTRS, PERMIT, ENY, USER.ID, NO. OF DIR. POS, CLUSTER);

VALUE NO. OF USERS IS 11 CLSTRS, REC.P.CLUSTER, NO. OF DIR. POS;

NO. OF DIR. POS IS BY VALUE ST DIR. SUP. 0.1 Y PESS IT UP.

LITERAL NO. OF USERS, H.CLSTRS, REC.P.CLUSTER, NO. OF DIR. POS:

LITERAL ARRAY NO. OF TERMS, NO. OF KNOs, DIR. PTRS:

STRING ARRAY TERM.KNO, DIR.KNO, USER.ID, CLUSTER:

BOOLEAN ARRAY PERMIT, DENY:

BEGIN CHECK:

LITERAL G.L. MASTER.THEITA, I, J, K, L, F, FILE.PTRS, AVG, N:

INTEGER ARRAY MASTER.THEITA(1:2), GAMMA(1:2, 1:50),

FILE.PTRS(1:2):

STRING ARRAY P.KNOs(1:2), AVG(1:14), FILE(1:1:1):

EXTERNAL BOOLEAN PROCEDURE, AGED:

EXTERNAL BOOLEAN PROCEDURE, ANAL:

EXTERNAL INTEGER PROCEDURE, DIR. SUR:

EXTERNAL PROCEDURE, CLUSTERS:

EXTERNAL INTEGER PROCEDURE, CLS. PTR:

G.W: LENGTH OF GAMMA:

L. MASTER.THEITA:

FOR I I UNTIL NO. OF TERMS DO:

FOR J I UNTIL NO. OF TERMS DO:

BEGIN:

EXTERNAL ARRAY G. THEITA(1:H.CLSTRS):

EXTERNAL I.THEITA, G.W, L, F, G, A, "MIN.P, SUB.K, CLS. SUR:

CLUSTERS(I, J, NO. OF KNOs(I, J), DIR. PTRS, DIR. KNOs, NO. OF DIR. KNOs, TERM.KNO, THEITA.L.THEITA, DENY):

END:

ADD ENTRIES TO GAMMA LIST:
! LOOP BY CLUSTERS IN INTERSECT.LIST (THETA):

FOR C_j UNTIL L.THETA DO BEGIN
  IF FIRST ADD THIS CLUSTER TO A "MASTER" LIST OF CLUSTERS IN WHICH RETRIEVALS WILL BE MADE:
    K_j;
    WHILE THETA[C_j]'MASTER.THETA[1] AND K_j<MASTER.THETA DO K_j+1;
    IF MASTER.THETA[K_j]'THETA[C_j] THEN
    IF NOT FIND THETA[C_j] ON "MASTER.THETA" AND IT AT END AND ADJUST LENGTH:
      BEGIN
        L.MASTER.THETA.L.'MASTER.THETA+1;
        MASTER.THETA.L.'MASTER.THETA+THETA[C_j];
      END;
  IF FIND KWD IN TEMP. WITH "IN P FOR THIS CLUSTER:
    M._P_REC.P.CLUSTER+1; INITIALIZED SO FIRST POS KWD WILL BECOME INIT KPRIME;
    FOR K_j UNTIL KWD.'NO.OF.KWD[1],J DO
      IF NOT (BOX[1,2](K_j[1],J,[)]) THEN
        BEGIN
          SUB_DIR.SUB_TEMP.K=SUB_DIR.SUB_TEMP.K+1,DIR.KSP.'NO.OF.DIR.KSP,DIR.PTRS);
          CLS.SUB_CLS.PTR(SUB,THETA[C_j],DIR.PTRS);
          IF DIR.PTRS(SUB,CLS.SUB+2)<"IN P THEN
            BEGIN
              SUB_DIR.P Dir.PTRS(SUB,CLS.SUB+2);
            END;
        END;
      END;
    END;
END;
IF we found a B-PHI K.N. SEE IF THE (KPRIME,C) PAIR IS ON GALMA.
IF NOT ADD THEM
IF A.L.P.<=REC.P.CLUSTER THEN
WE MUST HAVE FOUND A KPRIME. BECAUSE A.L.P IS OFF ITS INITIAL VALUE.
BEGIN
0;1;
WHILE 0<; AND NOT(THETA[C]=GAMMA[1,0] AND KK=GAMMA[2,0]) DO 0;0+1
IF THETA[C]=GAMMA[1,0] OR KK=GAMMA[2,0] THEN
DID NOT FIND (KPRIME,C) ON GAMMA. ADD TO END AND INCR LENGTH.
BEGIN
G:G+1;
GAMMA[1,G]_THETA[C];
GAMMA[2,G]_KK;
END;
END;
END;
BEGIN
SELECToutput (1).WRITE(" SEARCHING IN"):PRINT(L.\$ASTER.THERA,2);
WRITE(" CLUSTERS,\$1"): 
RETRIEVE RECORDS;
BEGIN
WRITE("NP\$"):PRINT(REC.P.CLUSTER);
FOR L_1 UNTIL L.\$ASTER.THERA DO
BEGIN
INTEGER ARRAY SIGNAL[1,REC.P.CLUSTER],SIG.FLG[1:REC.P.CLUSTER];
INTEGER SIG.\$HD,SIG.\$TR,FK,\$ME,\$KDS,\$UR,\$EL,\$K,\$Z,\$F,
EXTERNAL PROCEDURE \$ERG\$SIGMA;
EXTERNAL PROCEDURE GET.REC;
EXTERNAL PROCEDURE PROCEDURE SATISFY;
EXTERNAL PROCEDURE GET.VAR;
SIG.END;
K.PRIE.K.IDS[1];
CUR.CEL.*SIGA.NTH();
INITIALIZE SIGMA LIST FOR THIS CLUSTER;
OPEN.FILE(K, CLUSTER(CUR.CEL));
FOR II.I UNTIL G DO
13..;
IF CUR.CEL*GATHA(1, II) THEN
14.
BEGIN
15.
J.PRIE.K.IDS[1].PRIE.K.IDS[1];
K.GATHA(1, II);
P.*POS[1].PRIE.K.IDS[1].DIP.K.IDS[1];
K.CLS.PTR(K, CUR.CEL, MIN.PTRS);
HERE.SIGMA(01.PTS, I, K[1], SIGMA.SIGA_SIGA.SIGA.SIGA);
END;
19.
FOR SIG.PTR.I UNTIL SIG.END DO
20.
BEGIN
21.
IF SIG.FLG(SIG.PTR) THEN
22.
BEGIN
23.
SIG.FLG(SIG.PTR) 1;
24.
END
25.
END
26.
END
27.
BEGIN
28.
FOR Z.1 UNTIL K.FILE.PTRS DO
29.
BEGIN
30.
IF FILE.PTRS(Z) THEN
31.
BEGIN
32.
END
33.
END
34.
IF SA.FIND(0,PRIPE.KMS,P,KMS,FILE(Z)) THEN
  IF FILE(Z) IS A PRIME KMS "SEND ITS NO.-HILL PTR ONTO SIGMA1
  SEND.SIGMA(FILE.PTRS(Z),SIGMA.SIG,FILG.SIG.END)
END:
END;

I FIND USER CATEGORIES WHICH ARE SATISFIED BY THIS RECORD:

FOR II.1 UNTIL NO. OF USERS DO
  IF PERMIT(II.CUR.CEL) THEN
    BEGIN
      FLG_0: J_1;
      WHILE FLG_0 AND J<NO. OF TERM(S) 11 DO
        BEGIN
          IF SATISFY(FILE.OR,FILE.PTRS,TERM,KMS,NO. OF.KMS(I,I),II,II) THEN
            BEGIN
              FLG_1;
              SELECTOUTPUT(0);
              WRITE("(C)",TERM,")") PRINT(J, 3, 0);
              GIVE(USER.ID(II),FILE,AVP,II.FILE.PTRS,II.AVP);
            END:
            J_1+1;
          END:
        END;
    END;
  END;
END:
CLOSEFILE(3):
PROCEDURE CLUSTERS(I,J,K,KIDS,DIR_PTRS,DIR_KIDS,NO_OF_D,DIR_KIDS,TERM,KID,THETA,L,THETA,DERY);
VALUE I,J: INTEGER I,J,L,THETA,K,KIDS,NO_OF_D,DIR_KIDS;
INTEGER ARRAY THETA,DIR_PTRS;
BOOLEAN ARRAY DERY;
STRING ARRAY DIR_KIDS,TERM,KID;
BEGIN CHECK!

BOOLEAN CREFDED;
INTEGER U,SUR,H,H,X,SUR,SUB,",L,HEN,H,K,NH;
EXTERNAL BOOLEAN PROCEDURE SA,FIND;
EXTERNAL INTEGER PROCEDURE DIR_SUB;
EXTERNAL PROCEDURE SOUT;
EXTERNAL PROCEDURE SOUTH;
EXTERNAL BOOLEAN PROCEDURE JEGAN;
H,H,N (I,II) ! INITIALIZED AT ANY VALUE > # OF POSSIBLE CLUSTERS;
THIS ASSUMES THAT FIRST NO-NEG KID WILL INIT HIN.H,K,H,SUR AND II,SUB;
I FIND THE KEYWORD IN THE TERM WITH THE LEAST
NUMBER OF CLUSTERS;

FOR K, UNTIL U,KIDS DO
IF NOTCHECKED(TERM,KIDS(I,J,K)) THEN
ONLY WANT TO DO THIS FOR NON-NEG KIDS;
BEGIN
SUB_DIR_SUB(TERM,KIDS(I,J,K),DIR_KIDS,NO_OF_D,KIDS,DIR_PTRS);
BEGIN_DIR_PTRS_SUB(I,II);
IF HEN.H < 'IN.H THEN:
BEGIN
KID,SUB_K; H,SUB_SUB; HIN.H,HEN.H;
END;
END:
END;
I fill initial theta list in preparation for the intersection operation:

L.: -

FOR L#: 1 UNTIL DIR.PTRSUB[SUB,1] DO
BEGIN
  TAIL(L) = DIR.PTRSUB[SUB,1];
  N#: +3;
END;

L.THETA#: -1;
LDELETE DELETED CLUSTERS FROM CONSIDERATION;
FOR 0 up TO L.THETA DO
IF GUM[1,THETA(1)] THEN THA(0) - 1;
LDO THE TERMINAL INTERSECTIONS;

FOR K#: 1 UNTIL K#D.SU#B-1, K#D.SU#B+1 UNTIL N.KHDS DO
IF GERM[1, SUB, J,N, K] THEN
  FIELD ONLY ON THIS FOR NON-HER K#D;
BEGIN
  SUB.DIR.SUB(TERM.K#,1,J,X),DIR.KHDS,NO.DF.DIR.KHDS,DIR.PTRS);
  FOR I#: 1 UNTIL L.THETA DO
BEGIN
  IF META(I) = 1 THEN
  BEGIN
  TAIL(I) IN THE DIRECTORY OF THIS K#D;
  O#3:1, NOFOUND_TRUE - 2;
  FOR O#4 = 1 WHILE O#4 <= J#:1, P#RS[SUB,1] AND NOFOUND DO
  IF META(I) = DIR.PTRSUB[SUB,4] THEN NOFOUND_FALSE ELSE N#:4 + 3;
  IF NOFOUND THEN TAIL(I) = 1;
END;
END; OF K LOOP;
CLEAN UP THETA COMPRESS OUT NOI-SUCCESSFUL ENTRIES (-1-5);
SQUISH (THETA, L, THETA);
CHECKOFF 1;
END
PROCEDURE LISET(FULL, FILE, FILE, PTRS, II, FILE, PTRS, CLSTR, EXP, II, CE, II, OF, CLUSTERS, CLUSTER, TERM, KND, NO, OF, KNDS, IR, KND, PTRS, SEC, P, CLUSTER, II, OF, DIR, KND, AVP, II, AVP);

VALUE II, FILE, PTRS, II, AVP, SEC, P, CLUSTER;

INTER FILE, PTRS, II, AVP, SEC, P, CLUSTER, II, OF, DIR, KND, NO, OF, CLUSTERS;

LITSER ARRAY FILE, PTRS, II, CE, II, OF, CLUS, DIR, PTRS;

STLII ARRAY FILE, CLSTR, EXP, CLUSTER, TERM, KND, NO, OF, AVP;

BOOLEAN ARRAY FULL;

BEGIN;

CHECKEN I;

INTEGER I, J, I, CLSTR, SUB, CLS, INDEX, INS, ADDR, TEMP, L, THETA;

INTEGER ARRAY THETA[I:2:4]; NUMBER OF CLUSTERS;

BOOLEAN TRANS;

EXTERNAL INTEGER PROCEDURE DIR, SUB;

EXTERNAL INTEGER PROCEDURECLS, PTR;

EXTERNAL BOOLEAN PROCEDURE SATISFY;

EXTERNAL PROCEDURE CLUSTERS;

EXTERNAL PROCEDURE ROOT;

EXTERNAL BOOLEAN PROCEDURE SATISFY;

EXTERNAL PROCEDURE CLUSTER;

EXTERNAL PROCEDURE SPLIT;

EXTERNAL PROCEDURE CANDIDATE CLUSTERS;

EXTERNAL PROCEDURE群(II, NO, OF, KND, DIR, PFRS, DIR, KND, NO, OF, DIR, KND, TERM, KND, THETA, L, THETA, FULL);

FOR I = 1 UNTIL L, THETA DO;

BEGIN;

IF NOT (SATISFY(FILE, II, FILE, PTRS, CLSTR, EXP, II, CE(THETA[I]), I, THETA[I]))

THEN THETA[I] = 1;

END;

SOJISHK(THETA, L, THETA);

THIS EXPRESSSES OUT ALL THE -1'S;

TRANSR_FALSE;

IF L, THETA = 1 THEN

NEED A NEW CLUSTER, GET ITS NAME, ADD ITS CANONICAL EXPRESSION ETC.;
BEGIN
TRANSF_TRUE; ! BYPASS TRANSFILE BECAUSE THIS IS A NEW CLUSTER;
NO_CLUSTERS = NO_CLUSTERS +1;
SELECTOUTPUT();
SELECTOUTPUT();
WRITE("[FILE.EXT FOR NEW CLUSTER—NUMBER IS ");
PRINTNO_CLUSTERS,J(IS);MKOUTPUT;
READ(CLUSTER(NO_CLUSTERS));
THET(I,J) = NO_CLUSTERS; L_THETA = 1;
FOR J = 1 UNTIL NO_KNDS(J) DO
CLSTR.EXP(I,J,J) = TRUE; KNDS(J) = TRUE;
AC(J,J,J) = NO_CLUSTERS; NO_KNDS(J) = FALSE;
END;
INSERT INTO ANY REMAINING MEMBER OF THETA;
IS.CLSTR yereta[1];
Fi1.0 INSERT ADDR BY FINDING THE RECORD COUNT OF ONE OF THE SECURITY;
K.NDS. ALL THE SAME SEC.K.NDS ARE TRUE IN THE CLUSTER;
SUB_DIR.SUBTER(I,J,K,J,KD,NO_DIR.KNDS,DIR_PTRS);
CLS.INDEX_CLS_PTR(SUB,INS.CLSTR.DIR_PTRS);
INS_ADDR.REC.P.CLUSTER-INS_DIR_PTR(SUB,CLS.INDEX+1) ! REC.PER.CLUSTER-P(K1,INS_CLSTR);
IADD REV FULL CLUSTER IF THIS IS THE LAST RECORD TO BE INSERTED;
INS_ADDR = I THEN
BEGIN
FULL(I,INS.CLSTR_TRUE;
END;
INOT DO THE K.NDS AND UPDATE THE DIRECTORY;
FOR I_1 UNTIL n.FILE.PTRS DO
END;

THEN OUTPUT THE RECORD AND COPY THE REST OF THE FILE INTO IT:

IF THIS IS NEW CLUSTER OPEN IT DIRECTLY, DO NOT FOOL WITH TMP.TMP;

IF TRANSBP THEN OPENFILE(5, CLUSTER(INS.CLSTR)) ELSE OPENFILE(5, "TMP.TMP");

SELECTOUTPUT(5);

PRINT(INS.AVR, 3, 3); NEWLINE;

PRINT(FILE.PTRN, 3, 3); PRINT(AVP, 3, 3); NEWLINE;

FOR L_1 UNTIL n.FILE.PTRS DO

BEGIN

WRITE("**") WRITE(FILE[1]); WRITE("**");

PRINT(FILE.PTRN[1], 3, 3); NEWLINE;

END;

FOR L_1 UNTIL n.AVP DO

BEGIN

WRITE("**") WRITE(AVP[I]); WRITE("**"); NEWLINE;

END;

IF NOT(TRANSBP) THEN

BEGIN

OPENFILE(3, CLUSTER(INS.CLSTR));

SELECTINPUT(3);

TRANSFILE;

207
PROCEDURE DIR_SET(KIDS, PTRS, N);

STRJ0 ARRAY KIDS; INTEGER ARRAY PTRS; INTEGER N;

BEGIN CHECK=1;

INTEGER I, J, M;
OPENFILE(2, "DIR.DAT");
SELECTINPUT(2);
READ(I); // NUMBER OF DISTINCT KIDS;
FOR I:=1 UNTIL N DO
BEGIN
READ(KIDS[I]); READ(PTRS[I,1]);
M:=2;
FOR J:=1 UNTIL PTRS[I,1] DO
BEGIN
READ(PTRS[I,J]); READ(PTRS[I,J+1]); READ(PTRS[I,J+2]);
M:=M+3;
END;
END;
CLOSEFILE(2);
CHECKOFF 1;
END
LEVIA.ALG 12-FEB-75 12:37  PAGE 19

PROCEDURE DIRPUT(KADS,PTRS,1);  
STRING ARRAY KADS; INTEGER ARRAY PTRS; INTEGER N;  
BEGIN CHECKOFF 1;  
INTEGER I,J;  
OPENFILE(3,"DIR.DMP");  
SELECTOUTPUT(5);  
PRINT(1); WRITELINE:NUMBER OF DISTINCT KADS;  
FOR I=1 UNTIL N DO  
BEGIN  
WRITE(""");WRITE(KADS[I]);WRITE("""); PRINT(PTRS[I,1]); !KI AND h(K1);  
FOR J=1 UNTIL PTRS[I,1] DO  
BEGIN  
PRINT(PTRS[I,1]);PRINT(PTRS[I,1+1]);PRINT(PTRS[I,1+2]);NEWLINE;  
END;  
END;  
CLOSEFILE(5);  
CHECKOFF 1;  
END;
PROCEDURE CE.FILL (CLUSTERS, FULL, NO.CE, CLSTR.EXP, NO.OF.CLUSTERS):

INTEGER NO.OF.CLUSTERS; INTEGER ARRAY N.CE;

BOOLEAN ARRAY FULL;

STRING ARRAY CLUSTERS, CLSTR.EXP;

BEGIN CHECKOFF 1;

INTEGER I, J, FULL, I.Ed, EXP;

OPENFILE(1, "CLUSTERS.DAT"); SELECTINPUT(3);
READ(NO.OF.CLUSTERS);
FOR I:= 1 UNTIL NO.OF.CLUSTERS DO
BEGIN
READ(CLUSTERS[I]);
READ(FULL[I]);
FULL[I] IF FULL[I] = 0 THEN TRUE ELSE FALSE;
READ(I.Ed[I]);
FOR J:= 1 UNTIL N.CE[I] DO READ(CLSTR.EXP[I, I, J]);

END;
CLOSEFILE(3);
CHECKOFF 1;
END
PROCEDURE ERETRY(CLUSTERS, FULL, OCD, CLSTR, EXP, NO. OF CLUSTERS);

INTEGER NO. OF CLUSTERS; INTEGER ARRAY H. GC;

ARRAY ALL CLUSTERS, CLST. EXP;

BEGIN CHECK 1:

DEFINE I, J:

OPENFILE(5, "CLSTRS.HE."); SELECTOUTPUT(5):

PRINT(0. O. OF CLUSTERS, 3, 3) NEWLINE:

FOR I = 1 UNTIL NO. OF CLUSTERS DO

BEGIN

WRITE("\n") WRITE(CLUSTERS(I)) WRITE("\n") WRITE("\n")

IF FULL(I, I) THEN PRINT(0, J) ELSE PRINT(1, J):

H. GC[I, J] = 0:

PRINT(I, CE[I, J]):

SPACE(3):

FOR J = 1 UNTIL CE[I, J] DO

BEGIN:

WRITE("\n") WRITE("\n") WRITE(CLSTR, EXP[I, I, J]): WRITE("\n") WRITE("\n") SPACE(2):

END:

NEWLINE;

END:

CLOSEFILE(5):

CHECKOFF 1:

SELECTOUTPUT(3) WRITE("233D9(C)"):

END:
PROCEDURE FILL(A, FILE);
STRING FILE; INTEGER I; STRING ARRAY A;
BEGIN CHECKON I;
I: INTEGER I;
OPENFILE(3, FILE); SELECTINPUT(3);
READ(A(I));
I:= I;
WHILE A(I)"#" DO
BEGIN
I:= I+1;
READ(A(I));
END;
END.
INTEGER PROCEDURE DIR.SUB(TERM, DIR.KND, NO.OF.DIR.KNDS, DIR.PTRS);

INTEGER NO.OF.DIR.KNDS; STRING TERM;
STRING ARRAY DIR.KND; INTEGER ARRAY DIR.PTRS;

!PROCEDURE TO RETURN DIRECTORY ROW SUBSCRIPT OF TERM;

!IF TERM IS NOT FOUND IN THE DIRECTORY, IT IS ADDED AND NO.OF.DIR.KNDS, AND DIR.PTRS ARE ADJUSTED;

BEGIN CHECK ON !;

INTEGER A, FLG;
A,3; FLG,3;
WHILE A <= NO.OF.DIR.KNDS DO
BEGIN
A, A+1;
IF TERM = DIR.KND(A) THEN
BEGIN
BEGIN
DIR.SUB_A, FLG, A, NO.OF.DIR.KNDS+1;
END;
END;
END;

IF FLG = 1 THEN
BEGIN
!KND NOT FOUND ADD AN ENTRY FOR IT IN DIR.KND, AND INIT DIR.PTRS FOR NEW ENTRY;
NO.OF.DIR.KNDS, NO.OF.DIR.KNDS+1;
DIR.KND(1), NO.OF.DIR.KNDS, TERM;
DIR.SUB NO.OF.DIR.KNDS;
DIR.PTRS(1), NO.OF.DIR.KNDS, 1, 0;
END;
CHECKOFF !;
END;
PROCEDURE MERGE_SIGMA(ADDR, SIGMA, SIG.FLG, SIG.END); VALUE ADDR: INTEGER ADDR, SIG.END: INTEGER ARRAY SIGMA, SIG.FLG;
BEGIN
    INTEGER I, J;
    CHECK ON I;
    I := 1;
    WHILE SIGMA(I) < ADDR AND I <= SIG.END DO I := I + 1;
    WHEN WE EXIT THIS LOOP I POINTS TO WHERE TO INSERT ADDR;
    WE MAY HAVE TO MOVE STUFF DOWN TO DO THE INSERTION;
    IF SIGMA(I) > ADDR THEN
        BEGIN
            FOR J TO SIG.END+1 STEP -1 UNTIL I+1 DO
                BEGIN
                    SIGMA(J) := SIGMA(J-1);
                    SIG.FLG(J) := SIG.FLG(J-1);
                END;
                SIGMA(I).ADDR := ADDR;
                SIG.FLG(I) := 1;
                SIG.END := SIG.END + 1;
        END;
    CHECK OFF I;
END;
PROCEDURE GET.REC(REC.NO, FILE, FILE.Ptrs, AVP, N.KNDS, N.AVP);
VALUE REC.NO, INTEGER REC.NO, N.KNDS, N.AVP;
STRING ARRAY FILE.AVP*;
INTEGER ARRAY FILE.PTR*;
BEGIN CHECK 1;
INTEGER REC, 1, IDUMMY;
STRING DUMMY;
SELECT INPUT (3);
READ(REC);
WHILE REC<REC.NO AND NOT(IOCHAN(3) AND AS0100) DO
BEGIN
FOR I = 1 UNTIL N.KNDS DO
READ(DUMMY); READ(IDUMMY);
FOR I = 1 UNTIL N.AVP DO READ(DUMMY);
END;
IF REC>REC.NO THEN
BEGIN
WRITE("[1]***ERROR IN GET.REC. RECORD NOT FOUND WAS "); PRINT(REC.NO);
END;
ELSE
BEGIN
FOR I = 1 UNTIL N.KNDS DO
READ(N.KNDS); READ(N.AVP);
READ(FILE(I)); READ(FILE.PTR(I));
END;
END;
CHECKOFF 1;
END.
PROCEDURE GIVE(USER, FILE, AVP, N.KNDS, N.AVP);
STRING USER; STRING ARRAY FILE, AVP; INTEGER N.KNDS, N.AVP;
BEGIN CHECKON 1:
  INTEGER I:
  SELECTOUTPUT(0):
  WRITE("Record for user "); WRITE(USER); NEWLINE:
  FOR I_! UNTIL N.KNDS DO:
    BEGIN
      WRITE(FILE(I)); NEWLINE:
    END:
  FOR I_! UNTIL N.AVP DO
    BEGIN
      WRITE(AVP(I)); NEWLINE:
    END:
  CHECKOFF 1:
END:
INTEGER PROCEDURE CLS_PTR(SUB, CLSTR, DIR_PTRS);
VALUE SUB, CLSTR: INTEGER SUB, CLSTR; INTEGER ARRAY DIR_PTRS;
BEGIN
CHECKON 1;
INTEGER I;
I :=
WHILE I <= DIR_PTRS(SUB, 1) AND DIR_PTRS(SUB, 3*I-1) = CLSTR
DO I := I + 1;
IF I > DIR_PTRS(SUB, 1) THEN
BEGIN
I := DIR_PTRS(SUB, 1), DIR_PTRS(SUB, 1) + 1;
DIR_PTRS(SUB, 3*I-1) := CLSTR;
DIR_PTRS(SUB, 3*I-1) := 0;
DIR_PTRS(SUB, 3*I+1) := 0;
END;
CLS_PTR_3*I-1;
CHECKOFF 1;
END;
END.
procedure squish(theta, l theta);

integer array theta; integer l theta;

begin
    checkon 1;
    integer to from;
    to_j;
    for from _1 until l theta do
        if theta[from]»1 then
            begin
                to _to+1;
                if to»from then theta[to]»theta[from];
            end;
        l theta _to;
    checkoff 1;
end;

end;
BOOLEAN PROCEDURE SATISFY(FILE,N,FP,TERM,KND,NO.OF,KNDS,I,J);

STRING ARRAY FILE,TERM,KND;
INTEGER NO.OF.KNDS;
INTEGER N,FP,I,J;
BEGIN CHECKON I;

STRING TST;
EXTERNAL BOOLEAN PROCEDURE SA.FIND;
INTEGER N,K,L,H;
EXTERNAL BOOLEAN PROCEDURE NEGKND;
BOOLEAN LOOP;
H,J,K,G; LOOP_TRUE;

LOOP THROUGH THE KNDS OF TERM.KND. IF EACH OF THE POSITIVE AND NONE OF THE NEGATIVE KNDS IS FOUND IN FILE THEN SATISFY_TRUE, ELSE FALSE;

FOR K=K+1 WHILE K<=NO.OF.KNDS AND LOOP DO
BEGIN
IF NEGKND(TERM.KND[I,J,K]) THEN
BEGIN
N_LENGTH(TERM.KND[I,J,K],2)
TST_COPY(TERM.KND[I,J,K],2,1)
ISTRIP OF THE MINUS SIGN OF THE NEG KND, AND SEE IF IT IS NOT THERE;
IF NOT(SA.FIND(N,FP,FILE,TST)) THEN M+1 ELSE LOOP_FALSE;
IF TST HAS FOUND IN FILE, EXIT, BECAUSE THE QUERY CANNOT BE SATISFIED BY THIS RECORD;
END
ELSE
BEGIN
IF SA.FIND(N,FP,FILE,TERM.KND[I,J,K]) THEN M+1 ELSE LOOP_FALSE;
END
END IF M<NO.OF.KNDS THEN FALSE ELSE TRUE;
TRUE IF ALL THE KNDS PRESENT IF POSITIVE ABSENT IF NEGATED;
CHECKOFF 1;
END;
END.
PROCEDURE INIT_PHOTO(USER.ID,USER,PERMIT,DENY);

INTEGER H.USER; BOOLEAN ARRAY PERMIT,DENY; STRING ARRAY USER.ID;

BEGIN CHECKON 1;

INTEGER PAT.USERS,PAT.CLSTRS,I,J,K,L;
STRING TST.ID;
INTEGER ARRAY PRO[I:20];
OPENFILE(2,"PROT.DAT"); SELECTINPUT(2);
READ(PAT.USERS); READ(PAT.CLSTRS);
FOR I..I UNTIL H.USER DO
FOR J..J UNTIL PAT.CLSTRS DO
BEGIN
PERMIT[I,J].FALSE;
DENY[I,J].TRUE;
END;
FOR I..I UNTIL PAT.USERS DO
BEGIN
READ(TST.ID);
FOR J..J UNTIL PAT.CLSTRS DO READ(PRO[J]);
FOR K..K UNTIL H.USER DO
BEGIN
IF USER.ID(K)=TST.ID THEN
BEGIN
FOR L..L UNTIL PAT.CLSTRS DO
BEGIN
IF PRO[L]>0 THEN TRUE ELSE FALSE;
END;
END;
END;
END;
CLOSEFILE(2);
CHECKOFF 1;
END;
BOOLEAN PROCEDURE NEKHD(KHD); STRING KHD; BEGIN CHECKON I; STRING TST; TST_COPY(KHD, I); NEKHD_IF TST="=" THEN TRUE ELSE FALSE; CHECKOFF I; END;
BOOLEAN PROCEDURE SA.FIND(LIMIT,SA,TST);
STRING TST;STRING ARRAY SA; INTEGER LIMIT;
PROCEDURE TO FIND IF TST IS IN SA FOR SOME SUBSCRIPT<LIMIT;
BEGIN CHECKON I:
    INTEGER I;
    LIMIT;
    WHILE I<LIMIT AND SA[I]#TST DO I:=I+1;
    IF I<LIMIT OR SA[LIMIT]=TST THEN TRUE ELSE FALSE;
    MUST HANDLE SA[LIMIT] SEPARATELY TO AVOID OUT OF BOUNDS MSG IF NOT FOUND;
    CHECKOFF I;
END#
```
END
READ (USER. ID)\nWRITE (USER ID, ) (RESULTS. DUMP)
BEGIN
IF I = 1 THEN USER ID = 1
ELSE
READ (USER ID, ) (RESULTS. DUMP)
END IF
WRITE (USER ID, ) (RESULTS. DUMP)
```

REDA (USERID, ID)\nVALUES (USER ID, ) (RESULTS. DUMP)
BEGIN
IF I = 1 THEN USER ID = 1
ELSE
READ (USER ID, ) (RESULTS. DUMP)
END IF
WRITE (USER ID, ) (RESULTS. DUMP)
```
SPECS.ALG  12-FEB-75  12:39  PAGE 2

IF A.S="Y" THEN
BEGIN
WRITE("USE EXISTING PROTECTION SPEC FILE? (Y/N)->") ; SPECTRANPUT;
READ(ALS);
IF ALS="Y" THEN
BEGIN
WRITE("READ (FILE: NAME)=") ; SPECTRANPUT;
READ(FILE);
OPEN(FILE,1,FILE);
SELECTINPUT(1);
READ(SPECS);
FOR I=1 UNTIL N.SPECS DO
BEGIN
READ(U.SAS[I]);
FOR J=1 UNTIL N.U.SAS[I] DO
BEGIN
READ(U.SAS[I,J]);
END;
READ(SPECS[I,I,J]);
END;
READ(SPECS[I,I,J]);
READ(YES.NO[I]);
END;
CLOSEFILE(1);
END;
WRITE("READING OF SPEC FILE COMPLETE ASK USER FOR MORE SPECS:");
SELECTINPUT(0);
IF WE HAVE INPUT SPECS FROM TTY SAVE NEW SPEC FILE;
IF KJ.1 NE 1
BEGIN
  IF SPEC’S . SPEC’S K:
  THEN
    OPEN FILE (3, "SPEC’S,FIL.");
    SELECT INPUT (3);
    PRIM(J, SPEC’S, J);
    RE-FILE;
    FOR I, I UNTIL J, SPEC’S DO
    BEGIN
      PRIM(I, SPEC’S (I), J, SPACE (3):
      FOR J, I UNTIL J USR’S (I) DO
      BEGIN
        WRITE ("***"); WRITE (USR’S (I, J)); WRITE ("***");
      END;
      RE-FILE;
      PRIM(J, SPEC’S, DS(I), J, SPACE (2):
      FOR J, I UNTIL J SPEC’S, DS (I) DO
      BEGIN
        WRITE ("***"); WRITE (SPEC’S (I, J, J)); WRITE ("***");
      END;
      RE-FILE;
      PRIM(YES, NO (I), J);
      RE-FILE;
    END;
    CLOSE FILE (3);
END;
BEGIN
  PROCESS SPEC FILE AGAINST CLUSTERS, NEG;
1.01.013 OPENFILE(2,"CLSTES,IE=");  
1.02.013 SELECTIPUT(2);  
1.03.013 READ(I.CLSTES);  
1.04.013 FOR I_1 UNTIL N.CLSTES DO  
1.05.013 BEGIN  
1.06.013 READ(ClUTY); READ(INDIC); IF FILE NAME AND FULL/EMPTY, WE DON'T NEED THEM;  
1.07.013 READ(J.KNOS); IF THE CLUSTER EXPRESSION;  
1.08.013 FOR L_1 UNTIL J.KNOS DO READ(ICE(L));  
1.09.013 FOR J_1 UNTIL N.USERS DO  
1.10.013 BEGIN  
1.11.013 FOR K_1 UNTIL N.SPECS DO  
1.12.013 DOES THIS SPEC CONCERN USER J?;  
1.13.013 BEGIN  
1.15.013 IF L<=N.USERS[K] AND SATISFY(ICE,K.KNOS,SPECS,N.SPEC.KNDS[K],1,K) THEN  
1.16.013 BEGIN  
1.17.013 TST=YES,P(I,J);  
1.18.013 IF PRO(J,I)-TST >= 4 THEN PRO(J,1)-TST = TST;  
1.19.013 ELSE  
1.20.013 BEGIN  
1.21.013 THIS IS A TEST FOR CONSISTENT ASSIGNMENT OF PROTECTION;  
1.22.013 IF IT TESTS FOR EITHER PRO(J,I)=4 INDICATING NO ASSIGNMENT MADE YET;  
1.23.013 IF FOR PRO(J,I)=TST, THE PROTECTION ASSIGNMENT FOR THIS SPEC;  
1.24.013 SELECTIPUT(1);  
1.25.013 WRITE("INCONSISTENT ");PRINT(K,3);PRINT(J,3);PRINT(I,3);NEWLINE;  
1.26.013 PRO(J,1)=1; DENY THE ACCESSES FOR NON;  
1.27.013 END;
Cited References


Date, C.J. An Introduction to Database Systems, Addison-Wesley, (1975).

Digital Equipment Corporation, DECSYSTEM-10 Assembly Language Handbook, DEC-10-NRZC-D.


KaffN75 Kaffen, N.E. et al "Design of a Secure Data Base Management System with Generalized Access Capabilities," The Ohio State University Computer and Information Science Research Center Report (to
appear).


McCaE75 McCauley, E.J. and Manola, F.A. "File Partitioning and Record Placement in Attribute Based File Organizations," (Submitted for publication).


RothJ74 Rothnie, J.B. and Lozano, T. "Attribute Based File Organization in a Paged Memory Environment,"


