A HIERARCHICAL SINGLE-KEY-LOCK ACCESS CONTROL USING THE CHINESE REMAINDER THEOREM

By

KIM SIN LEE

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1988

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
December, 1991

A HIERARCHICAL SINGLE-KEY-LOCK ACCESS
CONTROL USING THE CHINESE
REMAINDER THEOREM

Thesis Approved :

Huizhu Lu
Thesis Adviser

William David Miller

J Chandler

Thomas C. Collins
Dean of the Graduate College

PREFACE

The key-lock-pair mechanism based on the Chinese remainder theorem was modified and implemented on the single-key-lock system. The single-key-lock system associates each subject(i.e., user) with a key and each object(i.e., file) with a lock.

The modification is inspired by Chang's method of key-lock-pair mechanism using the Chinese Remainder Theorem. In addition to using the key-lock-pair (KLP) mechanism based on the Chinese remainder theorem, we introduce a hierarchical key storage structure which not only implies the relationship between the subjects, but decreases the number of recalculations of keys substantially when objects are added or deleted. This hierarchical key storage structure also requires fewer files or lock numbers to be involved in the key calculation. It also reduces the verification time to $O(\log_2 n)$, instead of $O(\log_2 N)$ which the old SKL system needs. Morever, during the calculation of keys for the subjects, faster computation speed is achieved by using the modulus congruence of a D_j,

where $$D_j = \prod_{i=1}^{n} L_i \quad \text{for} \quad i \neq j \text{ and } j = 1, 2, \ldots, n$$

where L_i denotes the lock on the file i for i=1, 2, 3, . . . , n.

A simulation of the single-key-lock access control was perfomed on a Vax/Unix machine and time complexity of the key calculation was discussed.

I wish to express my sincere gratitude to the individuals who assisted me in this project and during my coursework at Oklahoma State

University. In particular, I wish to thank my major adviser, Dr. Huizhu Lu, for her intelligent guidance, inspiration, and invaluable aid. I am also very grateful to the other committee members, Dr. William D. Miller and Dr. John P. Chandler, for their advisement during the course of this work.

My deepest appreciation is extended to my mother, who provided constant support, moral encouragement, and understanding.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUTION

Protection System

Protection systems in a computing environment are developed to prevent information stored in a computer from being destroyed, altered, or even disclosed or copied without being detected. With various resources in a computing environment, there is always a need to ensure that each user or process uses system resources only in ways consistent with the stated policies of the system administrators. Research in protection systems continues to grow as more sensitive information is stored and processed by computers and transmitted over computer communication networks. As more small businesses and even personal home computer systems become part of larger networks, the security of individual data becomes a growing concern.

There are three major areas of computer protection in a computer system, namely, the external protection, interface protection and internal protection. External protection is concerned with physical access to the overall computer facility. While interface protection deals with the authentication of a user once a physical access to the computer becomes feasible; the internal protection deals with the control of access to the computing resources, and safeguarding of information [Rusby and Randell, 83]. This research thesis will examine only the internal protection mechanism in the computer system, particularly on the access control of file in an operating system or file server.

Problem and Research Objective

Problem

Most current operating systems and databases make use of a combination of user list directory and file access control list. This combination works great as far as user access control is concerned. However, each request made by the user requires the monitor to do a lot of searching for the correct file and verify the validity of the request.

Research Objective

This research project aims to improve the speed of user verification when an access request is made to reduce the storage requirements imposed by the current linked list problem. Since the arithmetic computation generally takes up less computer time as compared to searching time, this research thesis aims to take advantage of that. It uses a unique Key K_i to represent each user in the system and a system identification number L_j for each file, and only through the system verification of K_i mod L_j which gives the access right of a_{ij}, and the system decides on the legality of the access attempt.

Graham and Denning's Monitor Model

There are two main reasons of studying a model.

1. By studying the security model, we have references to guide us in the design and implementation of secured database and system. especially in the area of determining the secured policies of the system. Therefore, before going on to explain the actual research methodology and objective, it

is important to clarify the security model this research work closely relates to.

2. Only through studying the properties of the models can a secured system designer differentiate the essence of the model from other secondary functions the system is entitled to provide. Figure 1 on the appendix shows the organization of the Graham and Denning's monitor model.

The Graham and Denning's model was first introduced by Lampson [Lampson, 71] and later modified by Graham and Denning [Graham and Denning, 1972]. Their major work was on the expansion of the generic protection properties of the model. There are four basic elements of the model.

1. A set of subjects S_i where $0 < i < N$ and N is the number of users in the system.
2. A set of objects O_j where $0 < j < M$ and M is the number of files in the system.
3. A set of user defined access rights R.
4. A set of system stored Access Control Matrix A.

The Access Control Matrix has an attribute for each subject, which is identified as a row. It also has an attribute for each object and is recognized as a column in the Access Control Matrix. The content of each matrix A_{ij} is the access right R_{ij}. For each object O_j, where j is any file in the directory, a subject S_i designates an "owner" in A_{ij}, then S_i has absolute control over object O_j. For each subject S_i, if another subject S_h (where $h < j$), designates a controller attribute, then S_h has more rights than S_i. There are eight basic protection rights described in the model. These protection rights are issued by various subjects and are taken by the system as commands. The commands will have effects on other subjects and objects. They are as tabulated as in table 1.

TABLE I

BASIC PROTECTION RIGHTS

1 *Create object*:	This command allows the issued subject to introduce a new object into the system.
2 *Create subject*:	This command allows the issued subject to create another subject or directory in the system.
3 *Delete object*::	This command allows the issued subject to delete an unwanted object from the system.
4. *Delete subject*:	This command has the rights to delete some directory or any other subject under its hierarchy.
5 *Read access right*:	This command allows a subject to determine the current access rights of a subject to an object.
6 *Grant access right*:	This command allows the *owner* of an object to allow other subjects to have the access rights designated by him.
7 *Delete access right*:	This command allows a subject to delete a right of another subject for an object, provided that the deleting subject is either the owner of the object or controls the subject from which access should be

TABLE I(Continued)

8. *Transfer access rights:*	This command allows a subject to transfer one of its rights of objects to another subject. (Each right can be transferable or nontransferable. If a subject receives a transferable right, the subject can then transfer that right ---either transferable or not --- to other subjects. If a subject receives a nontransferable right, it can user the right, but cannot transfer that right to other subjects.). This set of eight rules provides the properties necessary to model access control mechanisms of a secured system.

This set of eight rules provides the properties necessary to model access control mechanisms of a secured system. Tabulated in Table II. is the Secured System Commands with various conditions and consequences when these commands are carried out.

TABLE II

GRAHAM AND DENNING'S SECURED SYSTEM COMMANDS

1 Command:	create object O_j
Condition:	nil
Consequence:	add column for object in A_{ij} place owner right in A[x,o].

TABLE II (Continued)

2 Command:	create subject s_i
Condition:	nil
Consequence:	add row for subject s in A_{ij}, place control in A[x,s]
3 Command:	delete object o_j
Condition:	owner of object o_j
Consequence	delete column j for subject i
4 Command:	delete subject s_i
Condition:	control in A[i,j]
Consequence:	delete row s_i
5 Command:	Subject s_i read access rights of object o_j
Condition:	Control subject s_i or owner of object o_j
Consequence:	Retrieve access rights A_{ij}.
6 Command:	Delete rights of s_i on o_j.
Condition:	Control subject s_i or owner of object o_j
Consequence:	remove access rights from A[i,j]
7.Command:	grants access rights r to s_i on o_j
Condition:	owner of o_j
Consequence:	add r to A[s,o]
8 Command:	transfer right r or r* from subject s to object o
Condition:	r* in A[x,o]
Consequence:	add r or r* to A[s,o]

The most important contributions this model towards the secured system are:

1. Each object has a unique identification number which is attached by the system to each access attempted by any subject.

2. Each and every attempted access by a subject to an object is validated by the system.

This research thesis closely follows Graham and Denning's model. In the implementation of the model, we assume that the files are the only objects protected by the system and the users in the system are the only subjects. The access rights of users towards the files constitute the access matrix A. This research thesis is implementing the basic protection rights in a Vax/Unix computer.

CHAPTER II

LITERATURE REVIEW

Current Protection System

It is the intention of every system administrator that every user can only be allowed to access those information files that he is authorized to access. When a user has intention of accessing any informational resources in any computing environment, the protocol that takes care of the file access control will verify the access requests issued by the user.

To date, most commercial and military computer systems make use of the access matrix to exercise their access control. The access matrix uses each row (i) to represent an accessor and uses each column (j) to represent the informational files. Each entry towards the access matrix (i, j) represents the access rights authorized [Graham and Denning, 1972].

The use of the access matrix is straight forward and simple where direct method is concerned. The most straight-forward way of implementing the access matrix is having a global two-dimensional array as a matrix table [Peterson and Silberschatz, 1983]. Each user of row (*i*) has a separate entry of access rights towards each file which is represented by a column(*j*). Figure 2.0 in the appendix shows the diagram for the access matrix table. However this system of protection has a problem when the system is large with numerous users and files in the system, the access matrix is sparse and the matrix table has to be kept in the auxiliary memory and therefore needs additional input and output [Pfleeger, 1989].

Capability System

In 1966, Dennis and Van Horn [1966] came out with an idea to solve the sparse matrix problem. They suggested using a linked list of users called Directory Access Control, in which each user has a separate entry of file identifiers and their corresponding access rights. There are both hardware and software implementations of this linked list of users' records [Figure 2.1]. The software implementation of this notion is to create a record for each user in the system. Each record contains various entries for the file or resources that a user is capable of accessing. Each entry for a file contains the name of the file, the access rights of the user on the file as well as a file pointer that tells the operating system the location of the file. The hardware implementation of this idea in inter-user protection called *capability* where each word in the memory is tagged with an extra bit. If the bit is off, then the word is an ordinary instruction or data, else the word can be loaded into the protection descriptor register.[Iliffe and Jodeit, 1962]. This particular tag architecture is called *a capability system* and it gives rise to two sets of data values and two sets of instructions, namely the ordinary data values in computation and protection descriptor values and ordinary instruction to load protection descriptor values. This system aims to differentiate the two sets of instructions and data values and prevent misprocessing of data values. Thus each user is provided with one segment as a record to store the capability or file pointer he is authorized to use. Each capability then contains separate read, write or execute permission bits so that different users have different access rights or capabilities towards the same files [Iliffe and Jodeit, 1962].

Though this capability system solved the problem of having the access rights implemented in a global table, it has many implementational disadvantages. One of them is the problem of revocation, namely, if user A allows user B to have the capability to read one of his files, he can not disable the file pointer or capability that user B has stored away somewhere in the computer memory. His only option is to destroy the original file, an action that affects other users who have the capability to access the same file [Kain and Landwehr, 1986]. The second disadvantage of the capability system is the problem of propagation that user B may copy the capability and distribute them to users to whom user A does not want the file to be exposed. There were certain controls that restricted the possibility of propagation, which the original capability system did not provide. These measures were devised to solve the problem of propagation and one of the example is using exhaustive searching for all users that have access towards the file. However, this requires X*Y number of sequential searchs for X numbers of users and an average of Y number of records in the system[Saltzer and Schroeder, 1975].

Propagation

Various implementational improvements in the mentioned constraints of the original capability system were proposed and tested. The CAP system [Needham, 1972] and Plessey 250 [England, 1974] assigned a *capability holding segment* to each user and only those segments were used to load and store capability information. In this way, other users could not make copies of the capability of the original user and propagation was prevented. Similarly the Burroughs B5000 family used the same concept in improving the capability by constraining the capabilities to be stored in the virtual processor stack and a table to prevent unauthorized access. Another approach in solving the propagation problem was having a depth counter set to a certain limit. Any access to the segment in order to obtain a capability to open a file caused the counter to increment by one; subsequently, any attempt greater than the limit generated an error by the operating system[Karger and Herbert, 1984]. These approaches in solving the propagation problem call for greater auditing and flexibility because any auditing and checking by the operation system required checking all users.

Revocation

In solving the revocation problem in a capability system, all access to a file has to go through an indirect file where it then retrieved the capability for the intended access file specified by the user. Only the file owner or the system administrator has the capability to destroy or change the indirect file, thus making revoking the access capabilities of the user possible[Redell 1974, Synder 1981, Wiseman 1986].

According to Saltzer and Schroeder [1975], the basic problem with a capability system is that the capability to access an object given by the object owner is analogous to having the owner gives the "ticket" for entry to the intended person; this "ticket" could be transferred freely without any independent control by the system. Therefore, their proposed method and implementation imposed limitations on copyability. This means extra precautions and resources at the expense of simplicity, flexibility and uniformity of capability as addresses.

Access Controlling List System

Instead of distributing a "ticket" for admission into the protected object like the capability system, each protected object in an access controlling list system has a separate file where all the user names and their corresponding access rights are presented. The operating system or the file server would verify any user who requests to access the protected object, by checking the user name in the access controller file of the object. The access controller contains the object pointer as well as the access control list. The access controller functions as an indirect access to the protected object; therefore, the access controller itself is protected against any user [Peterson and Silberschatz 1983, Downs 1985].

The use of an access controlling list system provides a last minute check on any attempt to access an object. It stops propagation by not only restricting the ability to copy and transfer, as does the capability system, but also by verifying every attempt to access any object. Revocation is more manageable because the owner of the protected object can just retrieve the access controller and change the names and their given access rights. This system of access control is illustrated in Figure 2.2 [Stoughton, 1981].

The access controlling list system no doubt has many benefits over the capability system, but it certainly has its implementational problem. According to Saltzer and Schroeder [1975], any attempt to access requires the system to go through several serial steps, such as accessing the pointer register to get entry into the access controller list to search for the proper access rights, and then accessing the object through addressing registers. Another disadvantage of the access lists system is, in a time sharing system, a complex mechanism is required to search and compare the names of users. This slows down the system. The third disadvantage is that the access controller list length varies for different objects, thus imposing some implementation problems requiring great care in the programming of the searching mechanism.

Shadow Register

The first disadvantage was solved by allowing an extra pointer register for each user as a *shadow register.* Each time a user issues a command to access a file, the indirect access controller copies the content (with file pointer and access rights) to the shadow register; thus subsequent access to the same file by the previous user goes directly to the extra register, saving some memory references. Revocability can only be rigidly preserved by having to clean all shadow registers and changing access rights [Swaminathan, 1985].

Group Divisions in Access Control List

The variable length of the access controller list and multiple users requiring lengthy search were solved by the method proposed by Ritchie and Thompson[1974] on a Unix system, where users are categorized into groups. Only three entries are allowed in the access controller list on each object: one entry for the object owner, one entry for the group and the last entry for all system users. .The price paid is inflexibility, because each object can only be accessed by a group. If more than one group need to access the object, it has to be placed as a public object.

Single-Key-Lock System

Though the Access Control list has solved some problems in the area of propagation, problems still remain in the areas of verification and revocation. In the area of revocation, any time a file owner wants to revoke another user's file access rights, the system needs to perform an exhaustive search in the access control list for the correct user. Only then is revocation possible. In the area of verification, if a user requests to access a file, the system needs to search for the correct file, then run an exhaustive search for the user's name in the access control list. Then the system retrieves the access rights which the file owner gave the user and compares them with the rights that the user would like to exercise. Thus each verification requires an exhaustive search which the Single-Key-Lock mechanism aims to avoid.

Single-Key-Lock System Using Vector Calculation

Based on the same concept prescribed by the previous two systems of access matrix, Wu and Hwang [1984] proposed a single-key-lock system using the Key-Lock-Pair (KLP) mechanism, where each user is system assigned a key and each protected object is assigned a system lock. The system will verify any request to exercise the access right on an object by *xth* user on *yth* protected object using a mechanism developed by Hwang and Ton [1980].

In this system, the key, lock and access rights are represented by numbers, and access is only permitted by the system when the access rights requested are less than or equal to the entries made in the matrix. The entries made in the matrix table are specifically given by the owner of the file. The locks are created based on the keys assigned by the system and the entries made by the user on the matrix. If K_i represents the *ith* user and L_j represents the *jth* file; then the access right of K_i on L_j is represented by a_{ij}. Through the calculation of $a_{ij} = K_i * L_j$ in the Galois Field (t) where t is the smallest prime number that is larger than all the access rights in the matrix table considered. Revocation based on new matrix entries only requires the system to recalculate the lock assigned to a protected object. The merit of this single-key-lock system lay in its simplicity and flexibility because of a single key and a single lock assigned to a user and a file as compared to the pointer method used by the capability system and access controller list system. Since the implementation of this system is protected in protection kernel like the monitor, it does not have any propagation and revocation problem. However, the single-key-lock system has a storage problem due to the length of its keys and locks. In 1989, Chang and Jiang [1989] improved on the current method by proposing the Binary Single-Key-Lock system, where the underlying matrix entries, keys and locks are represented in binary numbers; calculation of the keys and locks could therefore be done in simple logical AND and XOR operations. However, the binary single key lock system only solves the storage problem to a lesser extent; complex calculation of keys and locks still prevail.

Single-Key-Lock System Using the Chinese Remainder Theorem

Chang [1986] proposed a method using a concept similar to the Single-Key-Lock System proposed by Wu and Hwang [1984]. However, this method requires a system to assign coprime numbers to any new file in the system. Calculation of keys that represent the users' access rights are based on the coprime numbers. This method has a lesser storage problem which was restricted by the method described by Wu and Hwang. Therefore, instead of using $a_{ij} = K_i * L_j \quad GF(t)$, in the original Single-Key-Lock Pair mechanism, the calculation should make use of the Chinese Remainder Theorem with $a_{ij} = K_i \bmod L_j$ where t is the smallest prime number that is larger than all the access rights of the users. This mechanism of calculating the keys and locks is more efficient in terms of system assigned coprime numbers because, unlike the method proposed by Wu and Hwang [1980], which required an arbitrary nonsingular matrix of size m for m users in the system, the single key lock system based on the Chinese Remainder System only required an integer to represent the key. Where storage is concerned, Wu and Hwang's method [1980] needs *O(a.b)* where a is the number of users and b is the number of files. However, Chang's method requires only *O(a+b)* for each storage of key.

However, this mechanism that make use of the Chinese Remainder Theorem has its disadvantages too. One of the main disadvantage of this mechanism is the fact that the mechanism would have to recalculate all the keys of all users present in the system when a new file (or new coprime number) is being added to the system. If each calculation of a user in the system takes up t system time, then each new file being added to the system requires t*M system time if M number of users have account on this system.

CHAPTER III

RESEARCH PROCEDURES

Research Objective

Keeping in mind the benefits of the Single-Key-Lock system based on the Chinese Remainder Theorem in designing the protection protocol, this research aims to improve the speed of the system by incorporating both the simplicity of the Single-Key-Lock System based on the Chinese Remainder mechanism and the strict control the access control list commands. This research will exploit the compactness of the Single-Key-Lock pair mechanism where each new file is assigned a new pairwise coprime number. The access rights of any files will be incorporated into a legitimate user's key using the single-key-lock pair mechanism based on the Chinese Remainder Theorem.

Research Methodology

The method developed by this research will incorporate the user hierarchical system into the user structure. In this system, all subjects or users are arranged into a single hierarchical tree of directories. This hierarchy aims to provide a hierarchy of control of access, through the ability to modify the access rights of the subjects lower in hierarchy than the control subject. The use of this user hierarchy system makes it possible for the system to create a totally centralized control of all access decisions. For example, if a user adds a file into his system, only he has exclusive right to give access permission to other users in the system.

Each user node carries a local binary tree of records which contains information on each file the user has access right to. This information is restricted to the name of the file and the system assigned prime numbers only. The most important restriction of this system is that a user could only allowed to access file in his own directory. Any time a user request to access a file is generated, the system protection protocol will verify the legitimacy of the access right by searching for the file in his own directory. If the file name is right, then the system assigned prime number (which identify this file in the system) is retrieved. At the same time, the key of the user is also retrieved and the access rights could be verified by finding the modulus congruence of the key on the lock. Therefore, the records that store the information on each file are arranged in a local binary tree. The use of local binary tree is to facilitate the system in verifying the user access requests. Therefore, for each access requested by the user, we require a $\ln_2 N$ search for the file where N represents the number of files present in the local binary local directory.

The Hierarchical User Structure

After clearing the password file, each user would be given a record according to their login names and password. Each user node contains the following information:

1. A string to store the user name. This string is used to identify the user in the process.

2. A string to store the user's department. This string is used to identify the department the user belongs to. The department head has exclusive access rights to all the files his subordinates have.

3 A string to store the user's group that he is belongs to. This string is used to identify the user 's group for the system. The group leader also has exclusive access rights to all the files the group members have.

4. 64 bits to store the key of each user. Each time a new file is being added to the local binary file tree of the user, a new key is being issued by the system. The mechanism of calculating the key is based on the Chinese Remainder Theorem that will be discussed later.

5. 64 bits to store the value of L

$$\text{where } L = \prod_{k=1}^{n} L_k$$

and $1 \leq k \leq n$.

The value of L is put into the user record is to facilitate the calculation of the key when a new file is being added or deleted. Recursive function is used to traverse the local binary file tree. Therefore, each time this value is needed, it could be retrieved from the user node.

6. A local binary tree pointer that points to the head of the local binary tree. If there is a file being added or deleted, recalculation of the key of the user could be done by traversing the local binary tree file. Therefore, the head of each tree has to be placed in the user node.

<u>The Local Binary File Structure</u>

This local binary file structure contains all the information of the files that the user is accessable to. It has

1. A string of 20 characters to store the name of the file. This information is vital in searching for the correct file name during accessing, deleting and transferring of rights.
2. Thirty two bits to store the value of each file number that is assigned by the system. The values would be used to calculate **L** as above.
3. A file pointer that tells the location of the file in the system. If access request is being verified, the file pointer would direct the process in fulfilling the access request.

In this system, a new feature is also added to the Single-Key Secured System. Since each user node carries a local binary tree structure in his own directory, and those file present in the system are files that are accessible by the user. This design aims to shorten the verifying time where the coprime file number is needed to calculate the access right of the file with $a_{ij} = K_i \bmod L_j$. However, since a higher hierarchy node is designed to have exclusive access rights towards files of lower hierarchy, (but only to the extent of the same department or same group) there might be times a father node wants to access files of a son node and it happens that the file is owned by the son node. Therefore, in the local binary file directory of the father node, the file node is not found. Thus, a global binary tree that contains all the file present in the system does the job of fianl control. Each time a new file is being added to the system, the name of the file is being stored into the record and inserted into the global binary file structure.

The Global Binary File Structure

The global binary tree node contains the following information:

1. A string of 20 characters to store the name of the file.
2. An owner pointer that points to the owner of the file.

If a father node tries to access a file that belongs to his son node, then the system will verify it by searching for the file in his own local binary directory first. Since the file is owned by his subordinate, the file is not present in his own local binary directory. Then the system needs to perform the final check on the global binary tree. If the file is not found, then the file is definitely not present in the system. Otherwise, the owner pointer in the record points to the owner of the file (or user node). Information regarding the user's department and group is retrieved and compared with the accessor node information on department and group name. If the accessor node is found more superior than the owner of the file in terms of the user hierarchical structure, then the system allows the accessor exclusive access right towards the file. Otherwise, the file is not accessible by the accessor.

The Chinese Remainder Theorem

The research method requires the system to calculate the keys of each user by applying the Chinese Remainder Theorem. The Chinese Remainder Theorem states that:

Let $n_1, n_2, n_3, \ldots, n_r$ be positive integers such that $\gcd(n_i, n_j) = 1$ for $i \neq j$.

Then the system of congruences

$x \equiv a_1 \pmod{n_1}$
$x \equiv a_2 \pmod{n_2}$
$x \equiv a_3 \pmod{n_3}$

.
.
.

$x \equiv a_r \pmod{n_r}$

has a simultaneous solution, which is unique modulo $n_1n_2n_3n_4 \ldots n_r$.

Proof: We start by forming the product $n = n_1n_2n_3n_4 \ldots n_r$. For each $k = 1, 2, 3, \ldots r$, let

$$N_k = n/n_k = n_1 \ldots n_{k-1}n_{k+1} \ldots n_r;$$

in other words, N_k is the product of all the integers n_i with the factor n_k omitted. By hypothesis, the n_i are relatively prime in pairs, so that $\gcd(N_k, n_k) = 1$. According to the theory of a single linear congruence, it is therefore possible to solve the congruence $N_kx = 1 \pmod{n_k}$ call the unique solution x_k. Our aim is to prove that the integer

$$x = a_1N_1x_1 + a_2N_2x_2 + \ldots + a_rN_rx_r$$

is a simultaneous solution of the given system.

First, it is to be observed that $N_i = 0 \pmod{n_k}$ for $i \neq k$, since $n_k \mid N_i$ in this case. The result is that

$$x = a_1N_1x_1 + \ldots + a_rN_rx_r = a_k.N_k.x_k \pmod{n_k}$$

But the integer x_k was chosen to satisfy the congruence $N_kx = 1 \pmod{n_k}$, which forces the

$$x = a_k \cdot 1 = a_k \pmod{n_k}$$

This shows that a solution to the given system of congruences exits.(Adapted from Burton, 1976)

The uniqueness of the keys calculated using the Chinese Remainder Theorem should be absolute, so that confusion could not arise during the system verification of the keys to use the different access rights. Supposing two keys are found using the Chinese Remainder Theorem, and the $L_1, L_2, L_3 \ldots L_n$ represents the various files in the system created by the users.with $L_j > \max\{a_{ij}\}$ where a_{ij} represents the access rights of the users.K_i on L_j. And $D_j = L/L_j$ where $L = \prod_{k=1}^{n} L_k$

where $D_j.x_j = 1 \pmod{L_j}$ can be solved by using the Extended Euclidean Algorithm.

$$x = \sum D_j.\ x_j.\ a_{ij} \qquad (1)$$

$$y = \sum D_j.\ y_j.a_{ij} \qquad (2)$$

Clearly, $\quad D_j.x_j = D_j.y_j = 1 \pmod{L_j}$ for all j

$D_j(x_j - y_j) = 0 \pmod{L_j}$ for all j

therefore, $x_j = y_j \pmod{L_j}$ for all j ...(.3)

From (3) $x_j = y_j + M_j.L_j$ for some M

substituting $x_j = y_j + M_j.L_j$ into (1) we get $x = \sum D_j (y_j + M_j . L_j) a_{ij}$

$$x = \sum D_j y_j a_{ij} + \sum D_j a_{ij} M_j L_j$$

since $D_j . L_j = L$

$$x = y + L \sum M_j . a_{ij}$$

therefore, $x = y$

with this, the Chinese Remainder Theorem is proven.

Research Step

Application of the Chinese Remainder Theorem

This research will focus on the Chinese Remainder Theorem [Burton 1976] and developing an algorithm to implement the access control based on the idea discussed by [Chang 1986].

Finding the Coprime Numbers

This research will also develop and implement an algorithm to generate coprime numbers which would be assigned to the files as locks. The procedure that generates the coprime numbers should be protected from any users. The idea behind the calculation of coprime numbers is to get the first prime number in the natural numbers system 2 and the idea that any composite number can be divided by any prime numbers found in the algorithm and these prime number lies between 2 and the square root of the composite number. Therefore, in order to shorten the testing time, if the square of the testing prime number is greater or equal to the number being tested, then we can quit testing. Listed below is the algorithm on the finding of the coprime numbers.

```
1    firstprime <-- currentnum <-- 2  index <-- 0
2    for (i = 1; i < Maxprime; i++)
     begin
          success <-- FALSE
3         while ( success <> TRUE )
          begin

          currentnum <-- currentnum + 1;
4         for ( k <-- 0; k <= index; k <-- k + 1)
          begin
5              if ( ( currentnum mod prime[lastprimefound] ) = 0 )
               then stop;
6              if ((prime[lastprimefound])^2 >= currentnum ) then
               success <-- TRUE;

7              if ( success = TRUE ) then stop;
          end;
          end;
8    prime[i] <-- currentnum;
9    index <-- i;
     end;
```

These coprime numbers are going to be served as the unique identification number the system provide to the each individual file the system.

Calculation of Keys

With the result from above, a function to calculate the keys is developed and implemented. Each calculated key is kept in their respective user nodes. The user nodes are then arranged in a hierarchical form. The generation of user hierarchical would be based on the idea discussed by [Saltzer and Schroeder, 1976] and under the user hierarchical form, the users in the system is also divided into groups so that any revocation or introduction of new files into the group, only the group members is assigned a new key. Each group has a group administrator to take care of revocation and public file access rights. Calculation of the key is only dependent on the access rights of the public files as well as the access rights a group member towards any files in the same group.

Compared with the user hierarchical system proposed by Wu and Hwang[1984], this system has greater advantage because the Key Lock Pair mechanism has to solve a series of equation in order to find out the relationship of two users. Where h_{ij} is the relationship between two keys K_i and K_j, then the Keys of K_i and K_j could be found by the transpose of the m X m key matrix. Thus giving $h_{ij} = K_i * K_j$ for $1 \leq i \leq m$, $1 \leq j \leq m$. Conceptually, this method of assigning keys to the user is very similar to the *direct key assignment method* discussed by Chang and Jiang [1989]. However, their method has to go through a series of calculations to find out the relationship between two keys as well, thus increasing the system time. In this improved method of user hierarchical system, the relationship between two keys will be confirmed by checking immediately the hierarchical structure of the user. Comparison between the two key in the hierarchy should confirm the superior and inferior relationship between any two users. In terms of user extensibility, any new account given to any user means adding them in the user list in the system as well as in the appropriate hierarchy.

Modification of the Extended Euclidean's Algorithm

According to Chang's algorithm in solving the keys of the users, he proposed that:

If $L_1, L_2, L_3, \ldots L_n$ represents the files or locks numbers with $L_j > \max\{a_{ij}\}$,

where a_{ij} is the access rights of ith user on jth file. Then $L = \prod_{k=1}^{n} L_k$

and $D_j = L / L_j$. The equation of $D_j x_j = 1 \pmod{L_j}$ for $0 < x_j < L_j$, can

be solved (uniquely since $0 < x_j < L_j$) by means of the extended Euclidean Algorithm.

Greatest Common Divisors and Euclid's Algorithm

Definition :

Let any two numbers No and N1 be positive integers. A positive integer M is called a *greatest common divisor* of N_0 and N_1 and is denoted by GCD(N_0, N_1), if

1. M divides both N_0 and N_1, and
2. every divisor of both N_0 and N_1 divides M.

The *Euclid's Algorithm* for computing GCD(N_0, N_1) is to compute the *remainder sequence* $N_0, N_1, N_2, \ldots N_k$ where N_i, for $i \geq 2$, is the nonzero remainder resulting from the division of N_{i-2} by N_{i-1}, and where N_k divides N_{k-1} exactly (ie., $N_{k+1} = 0$). Then $GCD(N_0, N_1) = N_k$.

<u>Theorem 3.4</u>

The Euclid's Algorithm correctly computes $GCD(N_0, N_1)$.

Proof : The algorithm computes $N_{i+1} = N_{i-1} - Q_i N_i$ for $1 \le i < k$, where Q_i = Floor Value [N_{i-1}/ N_i]. Since $N_{i+1} < N_i$, the algorithm will clearly terminate. Moreover, any divisor of both N_{i-1} and N_i is a divisor of N_{i+1}, and any divisor of N_i and N_{i+1} is also a divisor of N_{i-1}. Hence **GCD**$(N_0, N_1) = GCD(N_0, N_1) = \dots = GCD(N_{k-1}, N_k)$.

Since $GCD(N_{k-1}, N_k)$ is clearly N_k, the algorithm is proved.

<u>Extension of the Euclidean Algorithm</u>

The Euclidean algorithm can also be extended to find not only the greatest common divisor of N_0 and N_1, but also to find integers X and Y such that $N_0X + N_1Y =$ **GCD**(N_0, N_1). The algorithm is as below:

<u>Extended Euclidean Algorithm</u>

begin

1 $X_0 \leftarrow 1;$

$Y_0 \leftarrow 0;$

$X_1 \leftarrow 0;$

$Y_1 \leftarrow 1;$

$i \leftarrow 1;$

2 **while** N_i does not divide N_{i-1} **do**

```
        begin
                Q <--- Floor Value [ N_{i-1} / N_i ];
                N_{i+1} <--- N_{i-1} - Q * N_i;
                X_{i+1} <--- X_{i-1} - Q * X_i;
                Y_{i+1} <--- Y_{i-1} - Q * Y_i;
                i <--- i + 1;
        end
    Return (N_i, X_i, Y_i);
    end
```

The worst case time complexity to find the integer GCD(a_0, a_1) is $O(\ln_2 5^{1/2} N)$ if $0 \le a_0, a_1 \le N$. [Knuth, 1980]

In solving the equation of $D_j x_j = 1 \pmod{L_j}$ for $0 < x_j < L_j$, we will be using modification of the Extended Euclidean algorithm which is faster and more efficient.

<u>Modification of the Extended Euclodean's Algorithm</u>

In order to improve the speed and overall system efficiency of the operating system, the extended Euclidean Algorithm that Chang [1986] suggested was working with large numbers that would take a longer time to solve for x_j in $D_j x_j = 1$ due to the tremendous number of equations when a large number of users are log onto the system.

for $D_j = L / L_j$ and

$L = \prod_{k=1}^{n} L_k$ where $L_1, L_2, L_3 \ldots L_n$ represents all the locks. In my opinion, the use of smaller numbers is possible. Instead of using D_j itself, the remainder of D_j when it is divided by L_j could also be used to solve for x_j. The following proof will indicate why:

Supposing $D_j^{*} = D_j \pmod{L_j}$ where $D_j = D_j^{*} + M_jL_j$ (some value of Mj)

Since $D_jx_j = 1 \pmod{L_j}$,

therefore, $(D_j^{*} + M_jL_j) x_j = 1 \pmod{L_j}$

$$D_j^{*}x_j + M_jL_j x_j = 1 \pmod{L_j} \quad \text{and}$$

$$D_j^{*}x_j = 1 \pmod{L_j} \quad \text{QED}$$

Therefore, there will be a procedure that will change the numbers to a modulus and then the extended Euclidean Algorithm will be applied. In the algorithm that finds the key of

the user is $K_i = \sum_{j=1}^{n} D_jx_ja_{ij} \mod L$

During the calculation of the keys, since $\sum_{j=1}^{n} D_jx_ja_{ij}$

in general is a large number compare to L. In order to avoid overflow in the calculation, we use the fact that

$$(a + b)(\bmod c) = \{ [a (\bmod c)] + b \} (\bmod c).$$ (Appendix B)

That is, when we are calculating the key, if the partial sum is greater than L, then the modulus of the partial sum will be obtained and used.

Algorithm on The Chinese Remainder Theorem

This algorithm determines the positive constant key K for a given n pairwise coprime locks L_i. and a corresponding set of access rights a_i.

Input : $L_1, L_2, L_3, \ldots, L_n$ and $a_1, a_2, a_3, \ldots, a_n$.

Output: K

1 **Read** L_i and a_i

2 **for** (num = i; num $\leq$ n; num<-- num + 1) **do**

$L = L * L_{num};$

3 **for** (num = i; num $\leq$ n; num <-- num + 1) **do**

$D_{num} = L / L_{num}$;

4 **for** (num = i; num $\leq$ n; num <-- num + 1) **do**

$\hat{D}_{num} = D_{num} \bmod L_{num};$

5 compute the x_j with $\hat{D}_{num}$ using the Extended Euclidean Algorithm.

6 **for** (num = i; num $\leq$ n; num <-- num + 1) **do**

$K = K + D_{num} * x_{num} * a_{num};$

7 **if** ($K > L$) **then** $k = k$ **mod** L;

8 **Return** K;

Various Binary Operations

The eight commands described by the Graham and Denning 's model are simulated to the closest.using the various binary operations of addition, multiplication and division. The idea is to simulate the Single-Key-Access-Control System using the Chinese Remainder Mechanism with improvement by having each user to have his or her own local binary file structure. The entire simulation is assumed to be simulated inside the Secured Kernel. The following listed are the binary arithmetic operations carried out in the simulation itself.

Binary Addition

Given a positional number system in base b = 2, the addition of two *n* digit positive numbers, the addend *x* and the augend *y* :

$x = (x_{n-1}, \ldots x_1, x_0), \quad y = (y_{n-1}, \ldots y_1, y_0)$

results in a sum $S = (S_n, S_{n-1}, \ldots S_1, S_0)$ where S_n can only take one of the two values 0 or 1 independently of b. When S_n is 1, it will often be considered as an overflow. Since in calculation of the Keys using the Chinese Remainder Theorem do not give rise to any negative numbers, therefore, it is not being considered as an overflow. The addition algorithmis expressed as below:

1. C_0 <-- 0 ($C_0$is the initial carry-in);

2. **For** i := 0 **Step** 1 **until** n-1 **do**

   begin

   S_i <--- (X_i + Y_i + C_i) **mod** b;

   C_{i+1} <-- Floor Value [(X_i + Y_i + C_i)/ b]

   end;

3 S_n <-- C_n;

Since $X_i + Y_i \leq 2(b-1)$ and the initial $C_0 = 0$, the maximum value for any C_i will be the Floor value of [2 (b -1) + 1/b] = 1.

Since this algorithm will examine every bit once, therefore, it is of

O (n) where n is the number of bits represented.

The Multiplication Algorithm

Given two n-digit positive integers, the multiplicand X and the multiplier Y, represented in a positional number system of redix b = 2.

$x = (x_{n-1}, \ldots x_1, x_0), \quad y = (y_{n-1}, \ldots y_1, y_0)$

their result is a 2n-digit positive numbers:

$R = (R_{2n-1}, R_{2n-2}, \ldots R_0)$ could be calculated by the following algorithm.

1 **Set** R_j <-- 0, for $0 < j < 2n$;

2 **For** i := 0 **Step** 1 **until** n - 1 do

3 **If** $y_i <> 0$ **then**

begin

    K <-- 0;

    For j := 0 **step** 1 **until** n-1 **do**

    begin

        t <-- $X_j * y_i + R_{i+j} + K$;

        R_{i+j} <-- t **mod** b;

        K <-- Floor value $[t / b]$

    end;

    R_{i+n} <-- K;

end;

Generally, if V and Z are the two numbers needed to be multiply, the upper bound of this algorithm is $O(\ln_2 V)$ or $O(\ln_2 Z)$ depending on which number is greater.

Division Operation of Two Positive Integers

The division operation has $(n + m)$ digit dividend x and an n-digit divisor y to produce two outputs, an $(m + 1)$-digit quotient q and and n-digit remainder r such that:

$$X = y * q + r, \qquad 0 < r < y$$

The above algorithm is called a restoring division with :

$$X = (X_{n-1}, \ldots, X_0)$$

$$y = (y_{n-1}, \ldots, y_0),$$

$$q = (q_n, \ldots, q_0),$$

$$r = (r_{n-1}, \ldots, r_0)$$

The algorithm can be expressed as :

1 Expand x into $x' = (x_{2n-2}, \ldots, x_n, x_{n-1}, \ldots, x_0)$

by letting all x_i, for $n \leq i \leq 2n-2$, be 0, (* perform a sign extension *)

2. **For** i:= 1 **step** 1 **until** n **do**

Set $z \leftarrow x' - 2^{n-i} * y$

if $z \geq 0$ **then** $q_{n+1-i} \leftarrow 1$ and $x' \leftarrow z$;

else $q_{n+1-i} \leftarrow 0$ and do not modify x'

3 r <-- x'

From the algorithm, it is clear that if the value of the number being divided is **V**, thus the upper bound of the binary division operation is in the $O(\ln_2 \mathbf{V})$.

Algorithm On Various Binary Tree Operations

Algorithm on Find Node

This algorithm is part of the operations on the Binary Tree. It receives the head of any binary tree, whether it is a global or local binary tree. A stack of pointer to tree nodes is being passed and this serves as a path on the searching direction. The found is served as a flag to indicate to the calling routine whether the node is found.

Input : Head, info, found, stack, stacktop.
Output: found, stacktop, Head;

```
1.  previousnode <-- head;
2   currentnode <-- head;
3   temp_top <-- -1;
4   temp_found <-- FALSE
5   WHILE (( temp_found <> TRUE) AND ( currentnode <> NIL))
    begin
        temp_top <-- temp_top + 1;
        temp_stack[temp_top] <-- currentnode;
        if ( currentnode->info = info) then temp_found <-- TRUE;
        else
            begin
```

```
                if ( currentnode->info < info) then
                     currentnode <-- currentnode->rightpt;
                else currentnode <-- currentnode->leftpt;
           end;
      end;
6     found <-- temp_found;
7     stack_top <-- temp_top;
8     copy ( temp_stack to stack);
9     Return (currentnode);
```

3.7.2 Algorithm on Modifying Tag of Tree Node

This algorithm will calculate the tag inside the stack of pointer to tree nodes. Since this is a height balance tree, on any particular tree node, the longest path to the right must not be more than one node length than the shortest path on the left of that particular node. If more than two is found, then the algorithm would stop and return the critical node.

```
      Input: head  (* head of the tree *)
           process  (* to differentiate Insertion and Deletion *)
           critical   (* an integer to indicate on stack which is critical *)
           stack (* an array of pointer to tree node *)
           stack_top (* an integer to tell top of stack *)
      Output:  critical node
1     previousnode <-- stack_top;
2     temp_top <-- stack_top -1;
3     temp_critical <-- FALSE;
4     STOP <-- FALSE;
5     Find critical loop:
```

```
if ( Deletion) AND ( stack[stack_top -1]->tag = 0) then
stop <-- TRUE;
if (stack[stack_top -1]->info > stack[stack_top]->info) then
begin
        if ( Insertion ) then
                decrement stack[stack_top-1]->tag by 1
        else    increment by 1;
end;
else begin
        if (Insertion) then
                Increment stack[stack_top -1]->tag by 1;
        else    decrement above by 1;
end
if ( |stack[stack_top-1]->tag| > 1 ) then
begin
        tempcriticalnode <-- stack_top -1;
        tempcritical <-- TRUE;
end;
if (( stop = TRUE) AND ( tempcritical = TRUE)
        OR  (stack[stack_top -1] = head)
        OR (stack[stack_top - 1]->tag = 0 )
        AND (Insetion))) then goto stopfind;
else begin
        previousnode <-- stack_top - 1;
        stack_top <-- stack_top - 1;
        goto Findcriticalnode;
end;
```

stopfind:

critical <-- temp_critical;

critical_node <-- temp_critical_node;

6 **Return;**

Algorithm On Binary Tree Insertion

This routine needs input on the head of the tree, the name of the file, and the pointer to the user node that tells who owns a file if inserton is done on the global binary tree. This routine also allocate memory for the new node being created for the binary tree (whether is local binary tree or global binary tree) as well as inserting the node into the lexicographic appropriate position. Before exiting the routine, it will call the balance tree routine to balance the tree after the new insertion.

Input: head (* either head for local or global binary tree *)

info (* name of the node *)

usernodept (* pointer to user node who owns the file *)

Ouput: head of the tree

1. location <-- **Call** findnode;

2 **if** (found = **FALSE**) **then**

3 **begin**

Allocate Memory for new node and update the information;

if (globalbinarytree) **then** newnode->ownerptr <-- usernode;

else newnode->filenumber = prime[primeindex];

end

4 **if** (head = **Nil) then** head <-- newnode;

```
5    if (location->info < info) then location->rightpt <-- newnode;
6    else location->leftpt <-- newnode;
7    Increment stack_top by 1;
8    stack[stack_top] <-- newnode;
9    Call ModifyTag;
10   if (critical) then Call BalanceTree;
     return;
```

<u>Algorithm On Deletion of Tree Nodes</u>

This routine will first search for the node in the tree according to the name of the file passed in. If the node is found, then it will delete the node from the tree and free the memory. After freeing the memory, it would then modify the tag on the path and if it is necessary, it will rebalance the tree. The output of this routine is the head of the tree.

Input: filename (* name of the file needs to be deleted *)

head (* head of the tree *)

Output: head of the node;

```
1    Call Find Node
2    if (found) then
     begin
         if (head node) then free (head);
         else if (head->rightpt = Nil) then
         begin
             head <-- head->leftpt;
             free(head->leftpt);
         end
```

```
else begin
    del_loc <-- stack_top;
    location <-- stack[del_loc]->rightpt;
    while (location <> Nil) do
    begin
        stack_top <-- stack_top + 1;
        stack[stack_top] <-- location;
        location <-- locatio->leftpt;
    end
    suc <-- stack_top;
    bef_suc <-- stack_top -1;
    Call ModifyTag;
    if (stack[del_loc]->rightpt = Nil) AND
        (stack[del_loc]->leftpt = Nil)) then
    begin
        if (stack[bef_del]->info > stack[del_loc]->info)
        then stack[bef_del]->leftpt <-- Nil;
        else stack[bef_del]->rightpt <-- Nil;
        free(stack[del_loc]);
    end
    else if (stack[del_loc]->rightpt = Nil) then
    begin
        if (stack[bef_del]->info > stack[del_loc])
        then stack[bef_del]->leftpt <-- Nil;
        else stack[bef_del]->rightpt <-- Nil;
        free(stack[del_loc]);
    end
```

```
                    else begin
                            copy (successor node to del_loc node);
                            if (bef_suc->info > suc->info) then
                            bef_del->leftpt <-- suc->rightpt;
                            else bef_del->rightpt <-- suc->rightpt;
                            free(suc);
                    end;
                    if (critical) then
                    begin
                            if (stack_top - critical_node) < 3) then
                            begin
                                    if ( critical_node->info >
                                            (critical_node +1)->info) then
                                    begin
                                            if (critical_node-.rightpt <> Nil)
                                            then begin
critical_node+ 1 <-- critical_node->rightpt
if (critical_node +1)->tag = 1 ) then
        critical_node+ 2 <-- critical_node->rightpt;
else if ((critical_node+1)->tag = -1 ) then
        (critical_node+2) <-- (critical_node+1)->leftpt;
else begin
                if ((critical_node +1)->rightpt <> Nil) then
                        critical_node+2 <-- (critical_node+1)->rightpt;
                else critical_node+2 <-- (critical_node+1)->leftpt;
end;
                                            end
```

end

end

Call BalanceTree

end;

end

Return

<u>Algorithm On Balance Tree</u>

This algorithm is called by the Insertion or Deletion routines. It receives input on the head of the tree, a flag to indicate Insertion or Deletion routine .and the stack where the path of all tree nodes are stored.

Input : head (* head of the tree node *)

flag (* to show Insertion or Deletion *)

stack (* the stack of tree node pointers for the path *)

critical_node (* node which is found critical *)

Output : head of the tree;

1 son <-- critical_node + 1;

2 grandson <-- critical_node + 2;

3 **if** ((stack[critical_node]->leftpt = stack[son]) **AND**

(stack[son]->leftpt = stack[grandson])) **then**

Call SingleLeftRotation;

else if ((stack[critical_node]->rightpt = stack[son]) **AND**

(stack[son]->rightpt = stack[grandson])) **then**

Call SingleRightRotation;

else if ((stack[critical_node]->leftpt = stack[son]) **AND**
(stack[son]->rightpt = stack[grandson])) **then**
Call DoubleLeftRotation;
else Call DoubleRightRotation;

4 **Return;**

Algorithm On Single Left Rotation

This routine is being called by the Balance Tree routine and the inputs include the head of the tree, stack that store the pointers of the path, and the critical node. This routine would bring the critical node down and put on the right of the pivotal node. It would then return the stack as well as the head of the tree.

Input: head (* head of the tree *)
stack (* stack that store pointers of the path *)
critical_node (* an integer that indicates the position of the critical node in the path *)

Output: head (* the head of the tree *)
stack (* the new stack with the nodes being repositioned *)

1 pivot <-- critical_node + 1;

2 pivot_right = stack[pivot]->rightpt;

3 stack[pivot]->rightpt <-- stack[critical_node];

4 stack[critical_node]->leftpt <-- pivot_right;

5 **if** (stack[critical_node] = head) **then** head <-- stack[pivot];

6 **else if** (stack[critical_node - 1]->leftpt = stack[critical_node]) **then**
stack[critical_node - 1]->leftpt = stack[pivot];

7 **else** stack[critical_node - 1]->rightpt <-- stack[pivot];

```
8       stack[critical_node]->tag <-- 0;
9       stack[pivot]->tag <-- 0;
        return;
```

Algorithm On Single Right Rotation

This routine will reposition the nodes in the path and takes input as head of the tree, the stack that store the pointers of the tree nodes as well as the position of the critical node. It would return the repositioned stack as well as the head of the tree.

Input: stack (* stack for the path pointers *)

head (* head of the tree *)

critical_node (* position of the critical node in the stack *)

Output: corrected stack, and the head of the tree.

```
1       pivot <-- critical_node + 1;
2       pivot_left <-- stack[pivot]->leftpt;
3       stack[pivot]->leftpt <-- stack[critical_node];
4       stack[critical_node]->rightpt <-- pivot_left;
5       if (stack[critical_node] = head ) then head <-- stack[pivot];
6       else if (stack[critical_node - 1]->leftpt = stack[critical_node]) then
                stack[critical_node - 1]->leftpt <-- stack[pivot];
7       else stack[critical_node - 1]->rightpt <-- stack[pivot];
        return;
```

Algorithm On the Double Left Rotation

This routine is called by the Balance Tree routine and takes input of head, stack and the critical node position. It would rotate once and then call

Single Left Rotation to do another rotation. Its output will be the stack and the head of the tree.

```
Input: head (* head of the tree *)
       stack (* stack that stores the pointers of the path *)
       critical_node (* position of stack that contains critical pointer *)
Output: head and the reposition stack;
1     pivot <-- critical_node + 1;
2     pivot_right <-- stack[pivot]->rightpt;
3     Copy input stack to local stack
4     stack[critical_node]->leftpt <-- pivot_right;
5     stack[pivot]->rightpt <-- pivot_right->leftpt;
6     pivot_right->leftpt <-- stack[pivot];
7     localstack[pivot] <-- pivot_right;
8     localstack[pivot+ 1] <-- stack[pivot];
9     Call Single Left Rotation;
10    if (( stack[critical_node]->rightpt <> Nil) AND
              (stack[critical_node]->leftpt = Nil)) then
              stack[critical_node]->tag <-- 1;
      else if (( stack[critical_node]->rightpt = Nil)  AND
              (stack[critical_node]->leftpt <> Nil))  then
              stack[critical_node->tag <--  -1;
      else stack[critical_node]->tag = 0;

11    if ((stack[pivot]->leftpt = Nil) AND ( stack[pivot]->rightpt <> Nil))
      then stack[pivot]->tag <-- 1;
      else if ((stack[pivot]->tag = 1) AND ( stack[pivot]->leftpt <> Nil)
              AND (stack[pivot]->leftpt->tag <> 0)) then
```

```
            stack[pivot]->tag <-- -1;
else if (( stack[pivot]->leftpt <> Nil) AND
        (stack[pivot]->rightpt = Nil)) then stack[pivot]->tag <-- -1;
else stack[pivot]->tag <-- -1;
return;
```

<u>Algorithm On the Double Right Rotation</u>

This routine is called by the Balance Tree routine and it takes input like the head of the tree, the stack that stores the path, and the critical node that indicates the position of the stack.

Input: head (* head of the tree *)

stack (* stack that stores the pointers of the path *)

critical_node (* position of stack that contains critical pointer *)

Output: head and the reposition stack;

```
pivot <-- critical_node + 1;
pivot_left <-- stack[pivot]->leftpt;
Copy input stack to local stack
stack[critical_node]->rightpt <-- pivot_left;
stack[pivot]->leftpt <-- pivot_left->rightpt;
pivot_left->rightpt <-- stack[pivot];
localstack[pivot] <-- pivot_left;
localstack[pivot+ 1] <-- stack[pivot];
Call Single Right Rotation;
if (( stack[critical_node]->rightpt <> Nil) AND
        (stack[critical_node]->leftpt = Nil)) then
        stack[critical_node]->tag <-- 1;
else if (( stack[critical_node]->rightpt = Nil)  AND
```

```
        (stack[critical_node]->leftpt <> Nil)) then
        stack[critical_node->tag <--  -1;
    else stack[critical_node]->tag = 0;

11  if ((stack[pivot]->leftpt = Nil) AND ( stack[pivot]->rightpt <> Nil))
    then stack[pivot]->tag <-- 1;
    else if ((stack[pivot]->tag = -1) AND ( stack[pivot]->rightpt <> Nil)
            AND (stack[pivot]->leftpt->tag <> 0)) then
                stack[pivot]->tag <-- -1;
    else if (( stack[pivot]->leftpt <> Nil) AND
            (stack[pivot]->rightpt = Nil)) then stack[pivot]->tag <-- -1;
    else stack[pivot]->tag <-- 0;
    return;
```

<u>Example On the Application</u>

In this example, we assume that there are a total of nine users in the system. The first user in the hierarchy is the system administrator, Sa and two department heads. Namely department A, Da and department B, Db. Department A has 3 users under his hierarchy. which are named as AU1, AU2, and AU3. On the other hand, deparment B has 3 users under his hierarchy and there are called BU1, BU2, and BU3 respectively. Figure 3.3 shows the hierarchical structure of the example system. The system administrator is charged wih the task of setting the accounts of different users in the system, and assigning the preliminary files to be used by each user. Supposing there are three library files, which was set up by the system administrator, which are named as LIB1, LIB2 and LIB3. The system administrator Sd decides that he would allow all users in department A to execute LIB1, LIB2, LIB3 and users in department B to read and execute LIB1, LIB2 and LIB3. Suppose that each user in the system decides to create a file of their own. Thus, representing :

Execute : 1

Read : 2

Write : 3 and

Own : 4.

If a user can read a file, then he has the right to execute also. If a user own a file, then he could execute, read and write on the file. Each time a file is created , the system will assign a new prime number to the file and insert it in the global binary directory. Thus, the prime number that represents each file in the system is as follows:

LIB1 = 5

LIB2 = 7

LIB3 = 11 (These are system files.owned by the system administrator Sd)

F1A = 13 (The first file belongs to department A)

F1B = 17 (The first file belongs to department B)

F1AU1 = 19 (The first file belongs to user 1 in department A)

F1AU2 = 23 (The first file belongs to user 2 in department A)

F1AU3 = 29 (The first file belongs to user 3 in department A)

F1BU1 = 31 (The first file belongs to user 1 in department B)

F1BU2 = 37 (The first file belongs to user 2 in department B)

F1BU3 = 41 (The first file belongs to user 3 in department B)

Calculation of Keys of Various Users

To calculate the keys of these user :

1. To calculate the key of the system administrator Sd, we have three files that are created by him in the system. There are LIB1, LIB2, and LIB3 with prime numbers 5, 7, 11 respectively. Since he owns all the three files, the access rights are 4 for these three files.

$$\text{Then } L = \prod_{k=1}^{n} L_k$$

and $D_j = L / L_j$. d_j is the remainder of D_j when it is divided by L_j. The equation of $d_j x_j = 1 \pmod{L_j}$ for $0 < x_j < L_j$, will be calculated. Therefore,

$L = 5.7.11 = 385$ and

$D_1 = 77$, $D_2 = 55$ and $D_3 = 35$.

$d_1 = 2$, $d_2 = 6$ and $d_3 = 2$

$x_1 = 3$, $x_2 = 6$ and $x_3 = 6$

Therefore, the value of the key is

$$(D_1 x_1 a_1 + D_2 x_2 a_2 + D_3 x_3 a_3) \bmod L$$
$$= (77(3)(4) + 55(6)(4) + 35(6)(4)) \bmod 385$$
$$= (924 + 1320 + 840) \bmod 385$$
$$= 4$$

2. The calculation of the key of department head A involves 4 files in his local binary directory. Since users in department A could execute LIB1, LIB2, and LIB3, his access rights on these files are 1 respectively. Department head A also has a file of his own, that is F1A and it has been assigned a prime number of 13. The calculation of key for department A is

as follows:

$L = (5)(7)(11)(13) = 5005$ and

$D_1 = 1001$, $D_2 = 715$, $D_3 = 455$, and . $D_4 = 385$

$d_1 = 1$, $d_2 = 1$, $d_3 = 4$ and $d_4 = 8$

$x_1 = 1$, $x_2 = 1$, $x_3 = 3$ and $x_4 = 5$

Therefore, the value of the key is

$= (D_1x_1a_1 + D_2x_2a_2 + D_3x_3a_3 + D_4x_4a_4) \bmod L$

$= \{1001(1)(1) + 715(1)(1) + 455(3)(1) + 385(5)(4)\} \bmod 5005$

$= 771$

3. The calculation of the key of department head B also involves 4 files in his local binary directory. Since users in department B, like department A could read and execute LIB1, LIB2, and LIB3, his access rights on these files are 2 respectively. Department head B also has a file of his own, ~~that~~ is F1B and it has been assigned a prime number of 17 The calculation of key for department B is as follows:

$L = (5)(7)(11)(17) = 6545$ and

$D_1 = 1309$, $D_2 = 935$, $D_3 = 595$, and . $D_4 = 385$

$d_1 = 4$, $d_2 = 4$, $d_3 = 1$ and $d_4 = 11$

$x_1 = 4$, $x_2 = 2$, $x_3 = 1$ and $x_4 = 14$

Therefore, the value of the key is

$=(D_1x_1a_1 + D_2x_2a_2 + D_3x_3a_3 + D_4x_4a_4) \bmod L$

$= \{1309(4)(2) + 935(2)(2) + 595(1)(2) + 385(14)(4)\} \bmod 6545$

$= 4237$

4. The calculation of user AU1, which is the first user inside department A. Besides having the access rights of 1 or execute on the LIB1, LIB2, and LIB3, it has its own file of F1AU1, which is given the prime number of 19 by the system. Therefore, the key is calculated as follows:

$L = (5)(7)(11)(19) = 7315$ and

$D_1 = 1463$, $D_2 = 1045$, $D_3 = 665$, and . $D_4 = 385$

$d_1 = 3$, $d_2 = 2$, $d_3 = 5$ and $d_4 = 5$

$x_1 = 2$, $x_2 = 4$, $x_3 = 9$ and $x_4 = 4$

Therefore, the value of the key is

$= \{D_1x_1a_1 + D_2x_2a_2 + D_3x_3a_3 + D_4x_4a_4\} \bmod L$

$= \{1463(2)(1) + 1045(4)(1) + 665(9)(1) + 385(4)(4)\} \bmod 7315$

$= 4621$

5. The calculation of user AU2, which is the second user inside department A. Besides having the access rights of 1 or execute on the LIB1, LIB2, and LIB3, it has its own file of F1AU2, which is given the prime number of 23 by the system. Therefore, the key is calculated as follows:

$L = (5)(7)(11)(23) = 8855$ and

$D_1 = 1771$, $D_2 = 1265$, $D_3 = 805$, and . $D_4 = 385$

$d_1 = 1$, $d_2 = 5$, $d_3 = 2$ and $d_4 = 17$

$x_1 = 1$, $x_2 = 3$, $x_3 = 6$ and $x_4 = 19$

Therefore, the value of the key is

$= \{D_1x_1a_1 + D_2x_2a_2 + D_3x_3a_3 + D_4x_4a_4\} \bmod L$

$= \{1771(1)(1) + 1265(3)(1) + 805(6)(1) + 385(19)(4)\} \bmod 8855$

$= 4236$

6.The calculation of user AU3, which is the third user inside department A. Besides having the access rights of 1 or execute on the LIB1, LIB2, and LIB3, it also has its own file of F1AU3, which is given the prime number of 29 by the system. Therefore, the key is calculated as follows:

$L = (5)(7)(11)(29) = 11165$ and

$D_1 = 2233$, $D_2 = 1595$, $D_3 = 1015$, and . $D_4 = 385$

$d_1 = 3$, $d_2 = 6$, $d_3 = 3$ and $d_4 = 8$

$x_1 = 2$, $x_2 = 6$, $x_3 = 4$ and $x_4 = 11$

Therefore, the value of the key is

$= \{D_1x_1a_1 + D_2x_2a_2 + D_3x_3a_3 + D_4x_4a_4\} \bmod L$

$= \{2233(2)(1) + 1595(6)(1) + 1015(4)(1) + 385(11)(4)\} \bmod 11165$

$= 1541$

7.The calculation of user BU1, which is the first user inside department B. Besides having the access rights of 2 or read and execute on the LIB1, LIB2, and LIB3, it also has its own file of F1BU1, which is given the prime number of 31 by the system. Therefore, the key is calculated as follows:

$L = (5)(7)(11)(31) = 11935$and

$D_1 = 2387$, $D_2 = 1705$, $D_3 = 1085$, and . $D_4 = 385$

$d_1 = 2$, $d_2 = 4$, $d_3 = 7$ and $d_4 = 13$

$x_1 = 3$, $x_2 = 2$, $x_3 = 8$ and $x_4 = 12$

Therefore, the value of the key is

$= \{D_1x_1a_1 + D_2x_2a_2 + D_3x_3a_3 + D_4x_4a_4\} \bmod L$

$= \{2387(3)(2) + 1705(2)(2) + 1085(8)(2) + 385(12)(4)\} \bmod 11935$

$= 9242$

8.The calculation of user BU2, which is the second user inside department B. Besides having the access rights of 2 or read and execute on the LIB1, LIB2, and LIB3, it also has its own file of F1BU2, which is given the prime number of 37 by the system. Therefore, the key is calculated as follows:

$L = (5)(7)(11)(37) = 14245$ and

$D_1 = 2849$, $D_2 = 2035$, $D_3 = 1295$, and . $D_4 = 385$

$d_1 = 4$, $d_2 = 5$, $d_3 = 8$ and $d_4 = 15$

$x_1 = 4$, $x_2 = 3$, $x_3 = 7$ and $x_4 = 5$

Therefore, the value of the key is

$= \{D_1x_1a_1 + D_2x_2a_2 + D_3x_3a_3 + D_4x_4a_4\} \bmod L$

$= \{2849(4)(2) + 2035(3)(2) + 1295(7)(2) + 385(5)(4)\} \bmod 14245$

$= 3852$

9.The calculation of user BU3, which is the third user inside department B. Besides having the access rights of 2 or read and execute on the LIB1, LIB2, and LIB3, it also has its own file of F1BU3, which is given the prime number of 41 by the system. Therefore, the key is calculated as follows:

$L = (5)(7)(11)(41) = 15785$ and

$D_1 = 3157$, $D_2 = 2255$, $D_3 = 1435$, and . $D_4 = 385$

$d_1 = 2$, $d_2 = 1$, $d_3 = 5$ and $d_4 = 16$

$x_1 = 3$, $x_2 = 1$, $x_3 = 9$ and $x_4 = 18$

Therefore, the value of the key is

$= \{D_1x_1a_1 + D_2x_2a_2 + D_3x_3a_3 + D_4x_4a_4\} \bmod L$

$= \{3157(3)(2) + 2255(1)(2) + 1435(9)(2) + 385(18)(4)\} \bmod 15785$

$= 13862$

CHAPTER IV

ANALYSIS OF RESEARCH RESULTS

Program Correctness

According to Graham and Dennig, it is necessary to prove the program and system correctness through two criteria:

1 Any request made by a user or subject K_i which leaves the protection state or the matrix $\mathbf{A}$ intact can not be an unauthorized access.

2. Any command made by a user or subject K_i which changes the protection state $\mathbf{A}$ can not lead to a new protection state in which some other users or subjects, such as K_m has unauthorized access to the same object L_j.

With respect to the first criteria, if the protection system is correct, the attachment of a unique key, which identifies the commanding subject to every request it makes, allows the protection system to identify the user and the file. It thus makes any reference easier and thus fulfills criteria 1. In another words, since both the Key and the Lock are unique, therefore, all requests are accountable.

The burden of proofs lies on the fact that the protection system calculates the unique key correctly, and the protection system interrogates the correct entry in the access matrix $\mathbf{A}$ and no other monitors except the secured protection system alters the contents of the access. Since no other mechanism alters the access rights of any file except the protection system, therefore, those files which are accessible to the user will only be presented during the calculation of the keys. Since the sets of access rights of any two users U_x and U_y are never the same (though they may have the same set of access rights to the same set of library files, as soon as one of them issues the command to create another new file, or is given a new access right to a new file, the key of the receiving user is not the same any more), therefore, the keys calculated are always unique.

With respect to the second criteria, the keys are only calculated based on the given access rights $\mathbf{a}_{ij}$:

$$K_i = \sum_{j=m}^{n} D_j . x_j . \mathbf{a}_{ij} \quad \text{mod} \quad L = \prod_{k=m}^{n} L_k$$

where $m \leq j \leq n$, $m \leq k \leq n$, and $m \leq n$ and the protection state of a file can be changed by a user, but the recalculation of the key is done by the mechanism in the protection system and posted to the user 's directory who received the new access right to a given file. If the access right is read, then the user can not change it to write because the key is being calculated and the user can not change the key.

Considering the classical problem of propagation and revocation mentioned widely in most methods. A department head H_0 allows a group of n staff members under him $S_0, \ldots, S_{n-1}, S_n$ to read a very important document of the department. Suppose further that H_0 intends that under no circumstances, should S_1 read this document. Under the access control list method and directory list method, the entry for this file could be revoked and deleted from the list. However, further provisions must be provided to prevent all other group members (from S_0, to S_n) copying this file indirectly to S_1. Using the improved method of calculating the keys and the locks, any user who does not receive this unique lock number in calculating his or her key, simply can not access that file because it is just not found in his own local directory. Thus, this method provides the possibility of having a policy in which only the owner of a file can have the power to grant access rights to others.

Time Complexity of the Chinese Remainder Theorem

Since the Chinese Remainder Theorem requires the following formula:

$$K_i = \sum_{i=0}^{m} D_j . x_j . a_{ij} \quad \text{mod} \qquad L = \prod_{j=0}^{s} L_j \;\ldots\ldots\ldots\ldots (1)$$

m : the number of users in the system.

K_i : the i user key in the system

D_j : is the product of all the relatively prime numbers except the *jth* prime

number. It is calculated from $D_j = \prod_{x=1}^{s} L_x \qquad x \neq j$,

L : is the final product of S relatively prime numbers or all the files in the system.

To deduce the time complexity of the Single Key Access Control using the Chinese Remainder Theorem mechanism, we need to look at the binary operations of the various components in the formula. Since there are S numbers of files in a local binary tree directory, if L_i represents a file number, then each binary multiplication needs $O(\ln_2 L_i)$. Therefore, to calculate the product of all L_i, where $i \leq n$, and n is the number of files in the local binary directory, we definitely need $(n - 1)O(\ln_2 L_i)$. To deduce the number of operations which are needed for each D_j, where $D_j = L/L_j$., D_j needs an operation of $O(\ln_2 L)$, since the number of operation depends on the greater number in the division., in this case, the product of L_i, L. Therefore, to get the total number of operations for all D_j, where $1 \leq j \leq n$, we need $nO(\ln_2 L)$. Therefore, total overall number of operations to calculate L and all D_j is

$$(n - 1)O(\ln_2 L_i) + nO(\ln_2 L)$$

Since the Chinese Remainder Mechanism requires for solving for x_j in this equation of $D_j x_j - 1 = M_j L_j$ (for some value of M_j) . . . .(1)

To find the time complexity of x_j, we started with equation (1), however; D_j could be written as $d_j L_j + e_j$ for some value of d_j. Thus, we have $e_j x_j - 1 = M_j\hat{\ } L_j$ (for some value of $M_j\hat{\ }$) and the time to convert D_j needs a modulus operation of $O(\ln_2 D_j)$, since finding x_j from $e_j x_j - 1 = M_j\hat{\ } L_j$ needs the most time of $O(\ln_2 5^{0.5} N) - 2$, (if $0 \leq x_j$, $L_j < N$).thus we have the time to find the e_j . Thus the entire operation of finding a single key is

$$(n - 1)O(\ln_2 L_j) + nO(\ln_2 L) + O(\ln_2 L/L_j) + O(\ln_2 5^{0.5} N) - 2$$

From (1), we know that the entire Chinese Remainder Theorem mechanism costs an upper bound of

$$nO(\ln_2 L)$$

Comparison of the Improved Methods With the Key-Lock-Pair Mechanism

The Key-Lock-Pair (KLP) mechanism based on the Chinese Remainder Theorem proposed by Chang requires the system to fetch for a lock of the corresponding file. This unique lock number is required to perform a mathematical operation of K_i mod L_j where K_i is the key number of user i. If we assume all locks are stored in a binary tree and the total number of files present in the system is N. Thus, to verify a user access right to a file the number of searching is $\ln_2 N$. The system also needs to perform the above K_i mod L_j operation. Therefore, total number of operation is $\ln_2 N + 1$.

The key of each user K is calculated based on the Chinese Remainder Theorem. If we represent jth file in the system by a unique number, L_j, then the key for ith user is calculated using the access right a_{ij} of the user to jth file. Then $D_j =$ is L/ L_j and x_j can be found by solving $D_j x_j - 1 = M_j L_j$ (for some value of M_j) by using the extended Euclidean's Algorithm. Since $0 \leq a_{ij} \leq 4$ with

0 = No access

1 = Execute

2 = Read

3 = Write

4 = Own

and $1 \leq x_j \leq L_j$.

Disadvantages of the KLP mechanism

The main advantage of the KLP mechanism lies on its simplicity and its process during verification of users' access rights. From the introduction, if we assume that there are N files in the system, then the N lock numbers are stored in a binary tree. Each verification process in this KLP mechanism needs a log N search as well as one operation of K_i mod L_j to obtain a_{ij}.

However, if we assume M users in the system, this method has the following disadvantages and there can be observed as follows.

<u>M Keys Calculation After One File Addition.</u> Any addition of a new file by a user in the system requires the system to recalculate each user's key. Even though many users may not have any access right to that file and receives a zero for their access rights towards that file, we still require the unique lock number of the new file to recalculate all the keys in the system because D_j = is L/ L_j. With M users in the system, then we need to recalculate M times. If we denote T_c as the time required to calculate the key of one user, then there is a $\ln_2 N$ search for the right place to insert and $M*T_c$ for M users. This clearly takes up tremendous amount of system time to include all the lock numbers in the calculation.

<u>M Keys Calculation after One File Deletion.</u> As we can see from above, if any user in the system decides to delete a file in the system, since the corresponding lock number and the access right have to be removed from each key calculation, all the keys in the system would then need to be recalculated with M calculations. Thus, with the $\ln_2 N$ search for the right file to delete, then another $M*T_c$ to recalculate M keys after a file deletion.

<u>Long Search Time During Each User Verification.</u> When a user wants to access a file, the system needs to verify the legitimacy of the access request of the user. The user may issue a string for the file name. If we

assume that each file name and its corresponding unique lock number is stored in a binary tree, then we need to have a $\ln_2 N$ search for the lock number and then perform the verification by performing K_i mod L_j operation. Therefore, total time during user verification is $\ln_2 N + 1$. With a large number of users and numerous files in the system, the search for user verification takes up a lot of time.

Advantages of the Improved Method Over the KLP

One Key Recalculation During Each Insertion of File. Since each file is inserted into both the local binary tree of the owner as well as the system global binary tree, we need to search for the correct positions in both binary trees to insert the file. Thus, the improved method requires log n + log N searching if we assume there are n number of files in the local binary tree and N number of files in the global binary tree. Therefore, total time required to perform an insertion is $\ln_2 n + \ln_2 N + T_c$ instead of $\ln_2 N + M*T_c$ in the KLP mechanism.

One Key Recalculation During Each Deletion of File. When there is a deletion of file, it is the same case as the insertion and there is only one recalculation of key. Thus the total time is $\ln_2 n + \ln_2 N + T_{cd}$ as compare to the KLP mechanism which requires $\ln_2 N + M*T_{cd}$, if we denote T_{cd} as the time needed to recalculate the key after the file deletion.

Shorter Search Time for User Verification. In the research procedure, the analysis below shows it has shorter searching time during user verification.

1. Lowest Hierarchy Has $\ln_2 n + 1$ Time

In the user hierarchical nodes, it is reasonable to assume that there are more than 50% of the users in the lowest hierarchy of the system. For example, students account in the university is more than the faculty and administrative account. When users in the lowest hierarchy issue commands to access a certain file, they have only $\ln_2 n + 1$ number of operations. They could only search for files in their local binary tree where n is assumed to be the number of files in the local binary tree. The rule is that if they found the file in their local binary tree, the total time of operation is $\ln_2 n + 1$ where $\ln_2 n$ is the worst case searching time and perform a $K_i \bmod L_j$ operation. If that file is not found in the local binary tree, then that means the user can not access that particular file.

2. Higher Hierarchy

Since the node in the higher hierarchy comprised less than 50% of the system population, the node in the higher hierarchy requires log n + 1 operations if the accessed file is in the local binary tree of the user. If the accessed file belongs to the accessor's descendent, then the accessed file may not be found in the local binary tree, and the system needs to find that file in the global binary tree to find the owner of the accessed file. One more comparison is needed to determine the relationship between the accessor and the owner of the accessed file. The accessed file could only be accessed by the ascendent of the owner. Therefore, in the worst case analysis, the total number of operations is $\ln_2 n + \ln_2 N + 1$.if the accessed file is not found in the local binary tree of the accessor.

For nodes in this higher hierarchy, there is a possibility that the system may not find the file name in the local binary tree, then we denote

P as the percentage of finding the jth file in the ith user local binary tree.

(1-P) is the percentage that this file is not found in the local binary tree.

Therefore, the node in the higher hierarchy needs an operation of

$P(\ln_2 n + 1)$ if the file he wishes to access is found in his local binary tree.

$(1-P)\{\ln_2 n + \ln_2 N + 1\}$ if the file belongs to one of his descendents and thus

the file is not found in his local binary tree.

Therefore, if the total number of operations in the KLP mechanism is Y_{KLP} and the total number of operations in the improved method is Y_{NEW}. Then $Y_{KLP} = \ln_2 N + 1$ to verify a user status in accessing a file, where N is the total number of files present in the system. For the improved method,

$Y_{NEW} = P\{\ln_2 n + 1\} + (1-P)\{\ln_2 n + \ln_2 N + 1\}$ with

n = average number of files in the local binary tree,

N = total number of files in the global binary tree as the KLP mechanism.

simplifying, we have

$$Y_{NEW} = P\ln_2 n + P + \ln_2 n + \ln_2 N + 1 - P\ln_2 n - P\ln_2 N - P$$

$$\mathbf{Y_{NEW} = \ln_2 n + \ln_2 N + 1 - P\ln_2 N}$$

when the population comprises less than 50% of the system population, we need to prove that under normal circumstances, most users would access files that are legitimately accessible by them, thus under that assumption, P is close to 1. Since our handicap in this analysis is the difficulty in measuring P, or the probability of a user legitimately accessing a file, our justification is that when most users access their own file, the KLP mechanism has a higher number of operations than the improved method. If that is true, then

$$Y_{KLP} - Y_{NEW} > 0 \ldots (1)$$

Then $(\ln_2 N + 1) - (\ln_2 n + \ln_2 N + 1 - P\ln_2 N) > 0$

$$\Rightarrow \ln_2 N + 1 - \ln_2 n - \ln_2 N - 1 + P\ln_2 N > 0$$

$$\Rightarrow P\ln_2 N - \ln_2 n > 0$$

$$\Rightarrow P > \ln_2 n / \ln_2 N$$

The analysis is that, as long as P, the probability of a user legitimately accessing a file, is greater than $\ln_2 n/\ln_2 N$, the KLP mechanism has a longer verification time than the improved method. Since the value for $\ln_2 n/\ln_2 N$ is relatively small for a large database system, we conclude that, under normal circumstances, P is close to 1. Therefore, in this improved method, the user in the higher hierarchy also has a shorter total number of operations. In addition, user in the lowest hierarchy always has $\ln_2 n + 1$ total operations. Thus, overall, the improved method has a shorter verification time than the KLP mechanism.

CHAPTER V

SUMMARY OF RESEARCH THESIS

Summary

Secured system and secured database are essential for data accuracy and information integrity in modern computing environment. Therefore, when designing the operating system or database system, great effort and time must be devoted on considerations of having a secured system that is free from undetected and unverified access on any information files. A secured system must be able to provide the mechanism for both separation of all users information as well as sharing of certain sharable informational files; these mechanism must be robust and yet easy to use.

A system designer is charged with the duty of finding out what should be protected as well as understanding the environment the protection system is based on. Through studying models, the essential components of a system is identified, and the interactions between these components must be studied carefully in order to design an efficient system. This research project referenced the Graham-Denning Monitor model. Therefore, criteria of the model are followed and can be seen throughout the content of this research project. Since the model calls for the protection of objects in the system and thus requiring the separation of subjects and objects, the Chinese Remainder Theorem is used to implement the separation as well as the necessary verification upon attempted access. Various mathematical verifications were given on the mechanism to show that this mechanism works in accordance with the model criteria. Each user in the research project belongs to a node in the hierarchical structure.

Generally, the rule set up is that users in the lower hierarchy do not get more resources. In another words,they do not have more access rights towards a fix number of files or they have less library files that can be accessed. In the implementation of this research project, the keys represent the subjects and the files represent the objects to be protected. Any access of objects need a user's key to verify the access rights. This mechanism is performed in the protection system, which is ideally placed close to the hardware of the computing environment. Thus, in the implementation process, various binary operations were coded to show that the mechanism can be implemented close to the hardware as well as preserving the accuracy of the mechanism.

In the analysis of the research project, discussion is provided on the mechanism correctness by showing close affinity to the two basic assumptions.

Finally, the analysis shows the performance of the Chinese Remainder mechanism required a time of $O(\ln_2 L)$ where L represents the product of all the coprime numbers in a local binary tree. The research projcet shows that Single Key Lock mechanism could be done much faster in terms of key calculation, insertion of files, deletion of files and finally verification time.

Future Work

Further research could be geared towards faster performance of the mechanism by considering the faster multiplication of binary numbers. Calculation requirement of keys for users in the same functional group when a new file is introduced could be further improved using some other mathematical mechanism. The storage structure for the keys could be modified to splay tree instead of a height balanced tree if priority of the subjects could be determined.

BIBLIOGRAPHY

Burton D.M. (1976). *Elementary Number Theory,* Allyn and Bacon, Inc. New York.

Computer System Organization , (1973): The B5700/ B6700 Series. New York Academic Press.

Chang, C. K., & Jiang, T. M. (1989). "A Binary Single Key System for Access Control." *IEEE Trans. Computers,* vol. 38, No. 10.

Chang, C. C. (1986). "On the design of a key lock pair mechanism in information protection systems." *BIT,* 26 (4), 410-417.

Dennis, J. L., & Van Horn, (Mar.,1966), "Programming semantics for Multiprogrammed Computations," *Commun.* ACM, vol 9, 143-155.

Downs, D. et al. "Issues in Discreationary Access Control." *Proc. 1985 IEEE Symp. Security & Privacy,* IEEE Comput. Soc. 1985, pp. 208-218.

England, D. (Aug., 1974) "Capability Concept mechanism and Structure in System 250," *IRIA Int., Workshop Protection in Operating Systems,* 63-82.

Graham, G. S. & Denning, P. J. (1972). "Protection-Principles and Practice." *Proc. AFIPS SJCC,* 40, 417-429.

Hwang, T. Y. & Ton, J. C. (1980). "An access control mechanism for computer system resources," in *Proc. Int. Comput. Symp.*, Taipei, Republic of China.

Iliffe, J. & Jodeit, J. (Oct, 1962). "A dynamic storage allocation scheme," *Comput. J.*, vol 5, 200-209.

Kain, R., and Landwehr, C. " On Access Checking in Capability-Based Systems." *Proc. 1986 IEEE Symp. Security & Privacy*, IEEE Comput. Soc 1986, pp. 95-100.

Karger, P.,and Herbert, A. " An Augmented Capability Architecture to Support Lattice security and Traceability of Access. " *Proc. 1984 IEEE Symp. Security & Privacy* IEEE Comput Soc 1984, pp.2-12.

D. E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, Massachusetts (1973).

D. E. Knuth, *The Art of Computer Programming*, Vol.2: *Seminumerical Algorithms*, Second Edition, Addison-Wesley, Reading, Massachusetts (1980).

Needham, R. (1972). ":Protection systems and protection implementations," in *FJCC, AFIPS Conf. Proc.*, vol. 41, pt.1, 571-578.

Pettofrezzo, A. J. & Byrkit, D. R. (1970). *Element of Number Theory*, Allyn and Bacon, Inc.

Pfleeger, C. (1989) *Security in Computing*, Prentice-Hall, Inc.

Ritchie, D. & Thompson, K. (Jul., 1974). "The Unix time sharing system," *Commun.* ACM, vol. 17, 365-375.

Redell, D. (1974)."Naming and protection in extendible operating systems," Ph.D. dissertation, Univ. of Calif., Berkeley.

Rusby, I. and Randell, B. " A Distributed Secure System. " *Computer*, vol.16 n7 Jul.1983, pp. 55-67.

Saltzer, J. H., & Schroeder, M.D. (Sept. 1975). "The protection of information in computer systems." *Proc. IEEE*, 63 (9), 1278-1308.

Stonghtom, A. " Access Flow: A Proctection Model which integrates access Control & Information Flow. " *Proc. 1981 IEEE Symp. Security & Privacy*, IEEE Comput Soc 1981, pp 9-18.

Synder, L. "Formal Models of Capability-Based Protection Systems." *IEEE Trans. Comput.*, vol. 30 n3 Mar 1981, pp.172-181.

Swaminathan, K. " Negotiated Access Control . " *Proc 1985 IEEE Symp. Security & Privacy*, IEEE Comput Soc 1985, pp. 190-196.

Wiseman, S. " A Secure Capability Computer System." *Proc 1986 IEEE Symp. Security & Pravicy*, IEEE Comput Soc 1986, pp 86-94.

Wu, M. L., & Hwang, T. Y. (1984). "Access Control with single key lock." *IEEE Trans. on Software Eng.*, SE-10 (2), 185-191.

APPENDIXES

APPENDIX A

PROVE OF A COMPLETE RESIDUE SYSTEM MODULO M

A *If C is a complete residue system modulo m and (a, m) = 1, then the set*

$$C' = \{ax + b \mid x \in C\}$$

is a complete residue system modulo m.

PROOF: According to the definition of a complete residue system modulo m, each integer is congruent to one and only one of the members of the set. Assume that

$ax_1 + b \equiv ax_2 + b \pmod{m}$

for two members x_1 and x_2 of C. Then

$$ax_1 \equiv ax_2 \pmod{m}$$

Then $\qquad x_1 \equiv x_2 \pmod{m}$

since $(a,m) = 1$. However, this contradicts the hypothesis that x_1 and x_2 are members of C since no two members of a complete residue system modulo m are congruent. Hence.

$C' = \{ax + b \mid x \in C\}$

is a complete residue system modulo m.

APPENDIX A (Continued)

B *If $(a, m) = 1$, then the linear congruence $ax \equiv b \pmod{m}$ has exactly one unique solution (or incongruent solution).*

PROOF : Let C represents any complete residue system modulo m.

By the above theorem, the set $\{ax / x \in C\}$ is also a complete residue system modulo m. Therefore, there exists only one element $x_0 \in C$ such that ax_0 is congruent modulo m to a given integer b. Hence, the linear congruence $ax \equiv b \pmod{m}$, where $(a,m) = 1$, has exactly one incongruent solution $x \equiv x_0 \pmod{m}$

(Adapted from Pettofrezzo and Byrkit, 1970)

APPENDIX B

PROVE OF (a + b) mod c = { [a(mod c)] + b } mod c

To prove $(a + b) \bmod c = \{ [a \pmod c] + b \} \pmod c$

proof : $(a + b) \bmod c$

$= a \pmod c + b \pmod c$

$= [a \pmod c] \pmod c + b \pmod c$

$= [a \pmod c + b] \pmod c$ QED

APPENDIX C

FIGURES

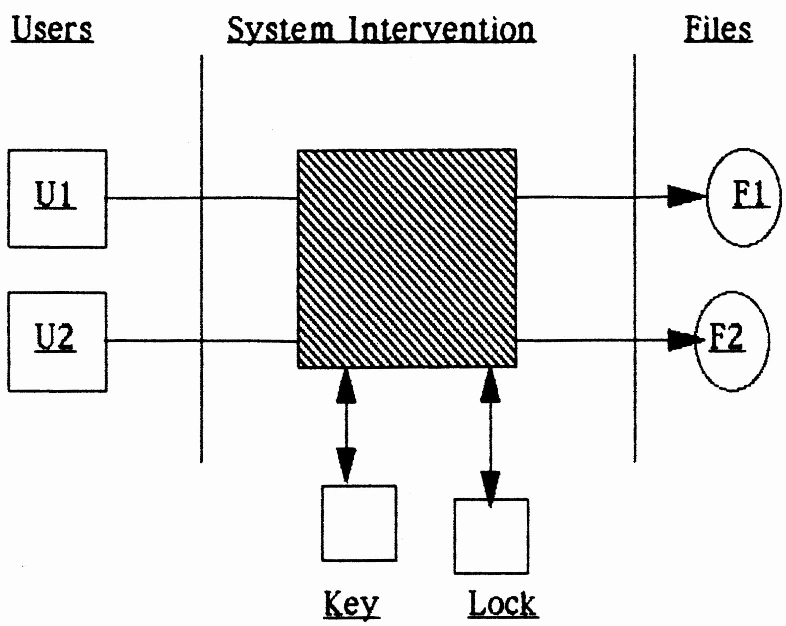


Figure 1. System View of SKL

	FILE 1	FILE 2	FILE 3	FILE 4	FILE 5	FILE 6	FILE 7
USER 1	O	R		R		R	E
USER 2	R	O					O
USER 3	R		R		R	R	R
USER 4	R						E
USER 5	W	R	R	E	E	E	E
USER 6	-	E	R	W			E
USER 7	-	-	-	R			E
USER 8	-	E	E	-	R	O	W
USER 9	-	E	E	W	-	R	W
USER 10	-	E	W	E	-	E	W

Figure 2.0. Access Control Matrix

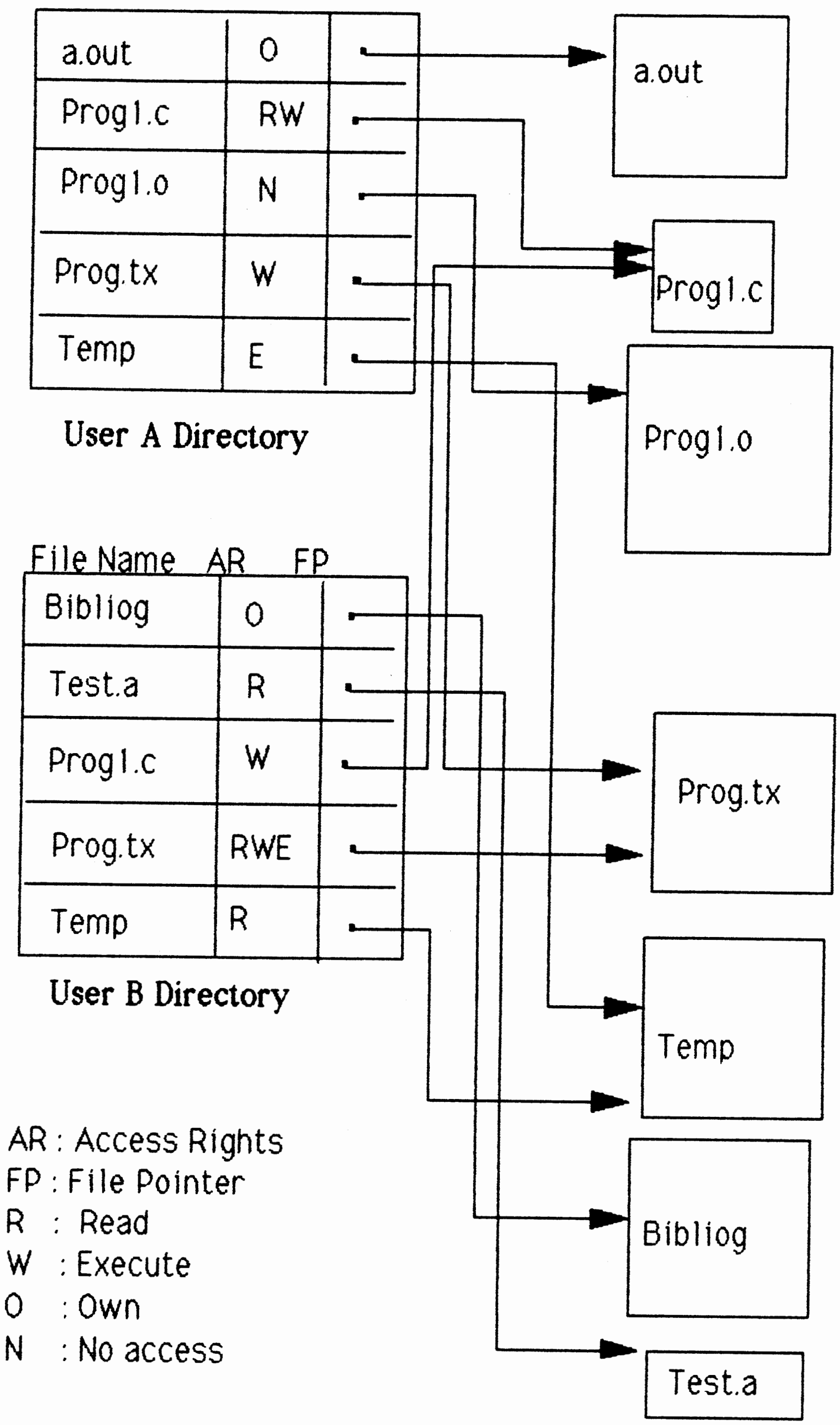


Figure 2.1. Directory Access Control

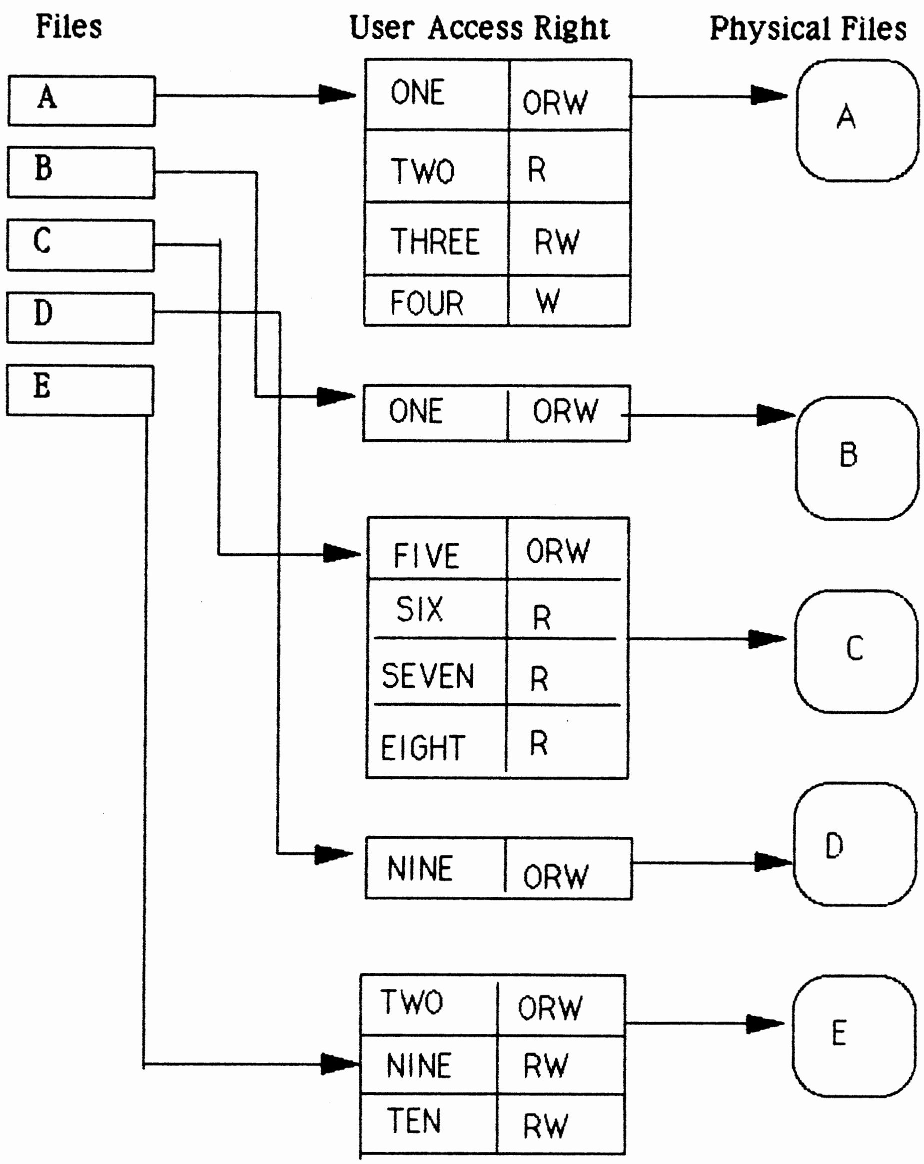


Figure 2.2. Access Control Lists

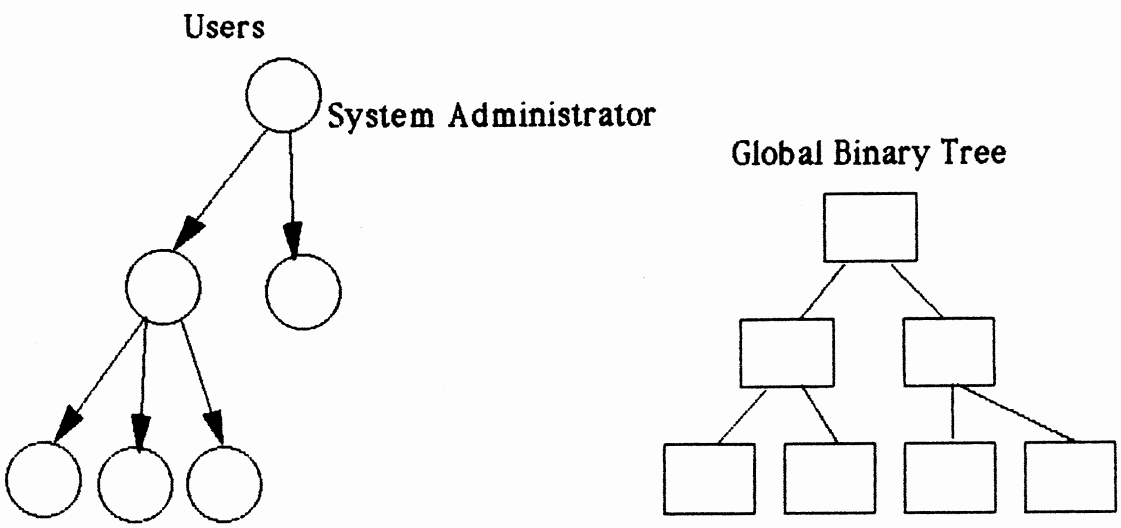


Figure 2.3. Structure of the Key-Lock-Pair Mechanism

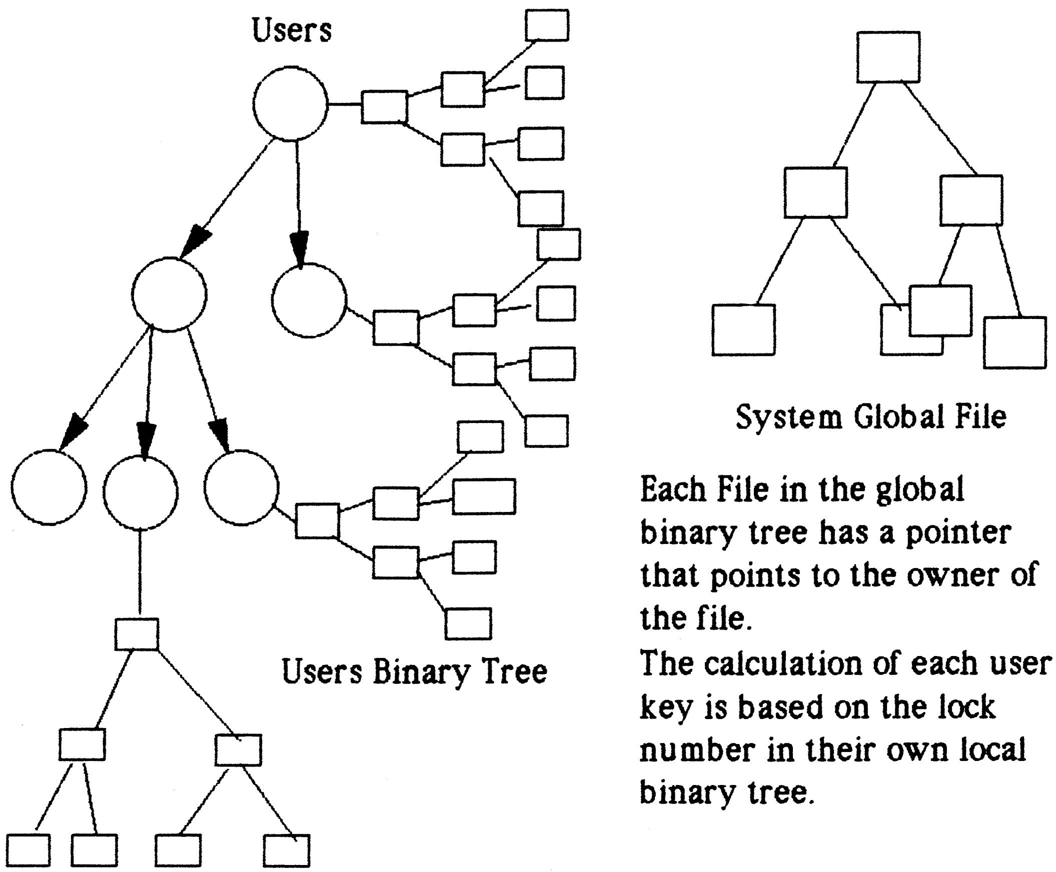
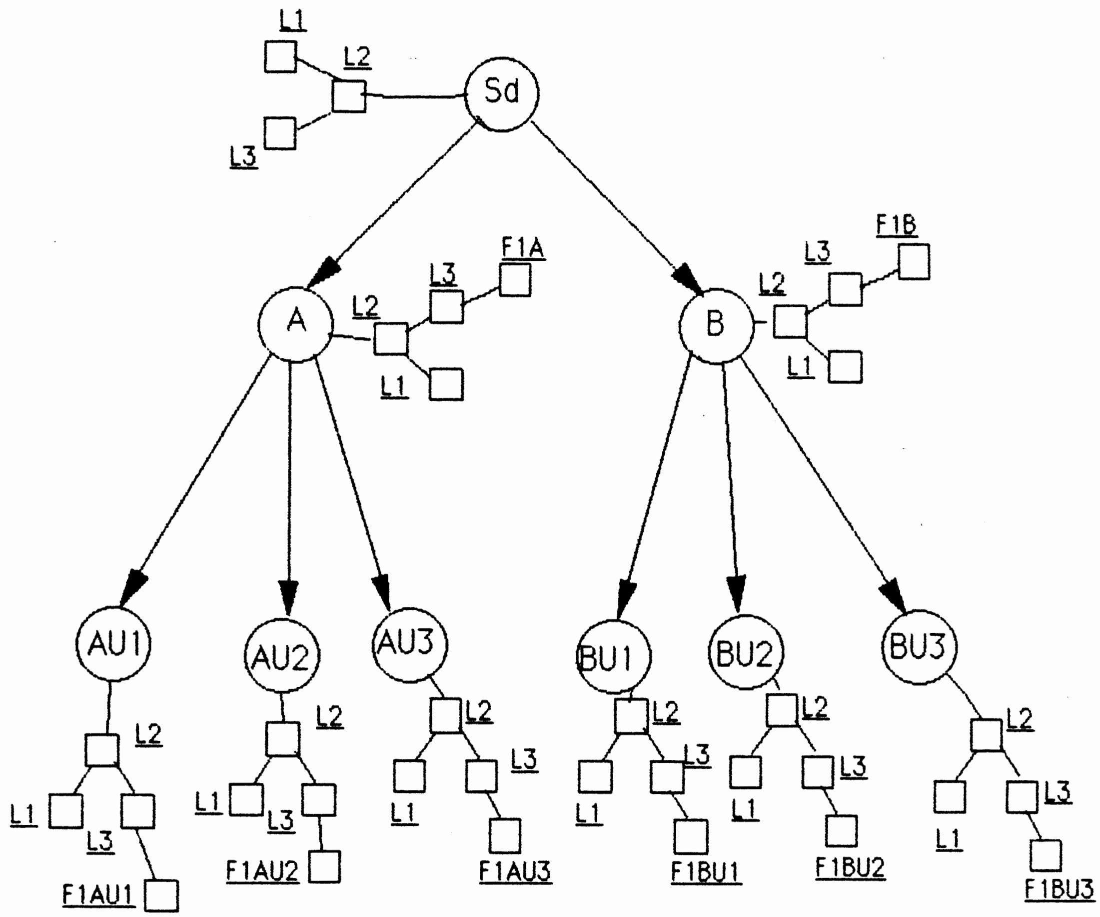


Figure 3.1 Hierarchical User Structure
With Local Binary Directory



L1, L2, L3 representing the 3 files by System Ad.
F1AU1 represents file owned by user 1 from
department A.

Figure 3.3. System View of Example File
Structure

APPENDIX D

SIMULATION OF A HIERARCHICAL SINGLE-KEY-LOCK ACCESS CONTROL USING THE CHINESE REMAINDER THEOREM

```
#include "header.h"
/*
  Name    : Lee, Kim Sin

  Tittle  : Computer Simulation on the Single Key Access Control using the
            Chinese Remainder Theorem.

  Project : Thesis Project for the Master of Science in Computer Science.

  Program Description : This program will simulate the Single Key Access
            Control using the Chinese Remainder Theorem. Each user in the
            computer system is given a node and they are being inserted into
            an ordinary hierarchical tree.

  Hierarchical Tree : Each user node contains information on the access rights
            of the user. A user is given some strings to identify the user
            himself as well as the department and the group that he belongs to.
            The key where the calculation is done is stored in the user node.
            The value of L where L represents the product of all the file
            number that is accessible by the user. Since the users on the same
            level have the same pririoty with the user himself, a pointer is
            inside the user node to let the process knows of the presence of
            other users. A pointer is also provided for users of lower level
            than him. A tree node is also provided to let the users have their
            files represented. Each file that is accessible by the user is
            being inserted in the local binary tree of the user. If the user
            wants to access a file, the operating system will check the
            legality of the request by retrieving the prime file number
            and retrieving the key of the user and perform the
            Key mod Lock = access rights. If the request is less than or
            equal to the access rights, then the request is granted.
            Else the request is not honor.

Binary tree : This binary tree will store all the necessary information on the
            file that is accessible by the user. Each node contains names
            of the file, the tag for rotation of the tree. A file number which
            is prime and represents the uniqueness of the file in the system. A pointer
            that points to the owner of the file. Two additional pointers that
            points to the right and left childen.

Logon On Structure: The function of this structure is to provide the process to
           recognize the user and passwords when they log on to the system, only
           recognizable password will be given access and directed to the
           correct usernode in the hierarchical structure.
*/
STRUCT tree_node_rec \{
        CHAR info[MAXLEN]; /*info is the file name */
        INT tag; /* to store the tag number of the file for easy balancing */
        INT fnum; /* to store the prime number associated with the file */
        STRUCT heirarchy    *ownerpt; /* a pointer that points to owner of file */
        STRUCT tree_node_rec *lpt; /*the left tree pointer */
        STRUCT tree_node_rec *rpt; /* the right tree pointer */
\} tree_node_type;
```

```
┌──STRUCT heirarchy \{
|       CHAR username[MAXLEN]; /* the user name in the system */
|       CHAR deptname[MAXLEN]; /* the department the user belongs to */
|       CHAR groupname[MAXLEN]; /* the group name the user belongs to */
|       CHAR key[MAX]; /* the value of the key in binary form */
|       CHAR large[MAX]; /* the value of all file number in the directory */
|       STRUCT heirarchy     *next; /* the next hierarchy pointer */
|       STRUCT heirarchy     *down; /* the subrodinate users in the system */
|       STRUCT tree_node_rec *head; /* the head of the local directory */
└──────────────\} heirarchy_entry;
┌──STRUCT logon \{
|       CHAR username[MAXLEN]; /* the user name in the system */
|       CHAR password[MAXLEN]; /* the password string belongs to the user */
|       STRUCT heirarchy *heirarchy_ptr; /* the pointer that points to heirarchy */
|       STRUCT logon *down; /* the down pointer */
└──────────────\} logon_entry;
  FILE *fp, *fg,*fch,*fl;
  INT   prime[Maxprime], primeindex, num;
  CHAR   first,second,str[80], globalkey[MAX], globallarge[MAX],
        dj[100][MAX], xj[100][MAX],aij[100][MAX];
  STRUCT heirarchy *h_start;
  STRUCT tree_node_rec *globalhead;
  STRUCT logon *logon_start,*logon_last;
  /*****************************************************************************
  Function Main : The function of the main program is to call various subsystems
        to facilitate the calculations of the key. It will call getprime() to
        produce a number of prime numbers which serves as the unique number
        when calculating the key. It also has a loop that will keep reading the
        input file for new users log on and new command issued. Thus, the
        emphasis of this program is on the batch processing of the various
        command.
  *****************************************************************************/
┌──main() \{
|    h_start=NULL;
|    logon_start=logon_last=NULL;
|    globalhead = NULL;
|
|    fp = fopen("userdata","r");
|   ┌──IF (!fp) \{ printf("can't open the input file \n");
|   |            exit(0);
|   └──\}
|    fg = fopen("globaltree.dat","w");
|   ┌──IF (!fg) \{ printf("can't open the write file \n");
|   |           exit(0);
|   └──\}
|    getprime();
|    primeindex =2;
|   ┌──WHILE(!feof(fp)) \{
|   |       ┌──IF (fgets(str,80,fp)) \{
|   |       |     IF (str[0] == '*') batch_process();
|   |       |     ELSE separate_string();
|   |       └──\}
|   └──\}
```

```
|   /*  search(h_start); */
|   /*   print_logon(logon_start); */
|     fclose(fg);
|     fclose(fp);
└───\}
   /*************************************************************************
   The getprime function will generate the prime numbers needed in the
   calculation of the keys and when which new file is being added into the
   system, the system will assign the new prime number for the file and this
   number will stay with the file for its entire life in the system. The
   prime numbers are stored in an array of integers and when ever there are
   needed, the system will fetch the number from the array.
   *************************************************************************/
   getprime()
┌───\{
| REGISTER i,k,success;
| INT current,index;
|
| prime[0] = current = 2;
| index = 0;
| ┌─FOR (i=1; i< Maxprime; i++) \{
| |      success = 0;
| |      ┌───WHILE (success != 1) \{
| |      |          current += 1;
| |      |          ┌───FOR (k =0; k <= index; k++) \{
| |      |          |       IF ( (current % prime[k]) == 0) BREAK;
| |      |          |       IF ( (prime[k] * prime[k]) >= current ) success = 1;
| |      |          |       IF (success == 1) BREAK;
| |      |          └───\}
| |      └───\}
| |      prime[i] = current;
| |      index = i;
| └──\}
| RETURN;
└───\}
   /*************************************************************************
   The separate string is called by the main function and it will separate
   the string that the main function sent into separate command that is
   recognizable by the system. Its primary function is to call various
   functions like the form_department, form_group, and form member with
   the commands issued in the batch file.
   *************************************************************************/
   separate_string()
┌───\{
|    CHAR s[80],name[MAXLEN],deptname[MAXLEN],groupname[MAXLEN],
|         password[MAXLEN];
|    REGISTER INT i,j,k;
|
|    i=j=0;
|    strcpy(s,str);
|
|    WHILE(s[i] != ' ') \{ name[j] = s[i]; i++; j++; \}
|    name[j] = '\0';
```

```
|
|   ┌──IF (s[i+1] == 's') \{
|   |     ┌──IF (s[i+2] == 'y') \{
|   |     |    j=0; i += 4;
|   |     |    WHILE (s[i] != '\0') \{ password[j] = s[i]; i++; j++; \}
|   |     |    password[j] = '\0';
|   |     |    form_sys(name,password);
|   |     └──\}
|   |     ┌──ELSE \{
|   |     |       j=0; i += 4;
|   |     |       WHILE(s[i] != ' ') \{ deptname[j] = s[i]; i++; j++; \}
|   |     |       deptname[j] = '\0';
|   |     |
|   |     |       i += 1; j =0;
|   |     |       WHILE(s[i] != ' ') \{ groupname[j] = s[i]; i++; j++; \}
|   |     |       groupname[j] = '\0';
|   |     |
|   |     |       i +=1; j =0;
|   |     |       WHILE(s[i] != '\0') \{ password[j] = s[i]; i++; j++; \}
|   |     |       password[j] = '\0';
|   |     |
|   |     |       form_member(name,deptname,groupname,password);
|   |     └──\}
|   └─\}
|  ┌──ELSE \{
|  |        k = i + 1;
|  |        i += 4; j =0;
|  |        WHILE (s[i] != ' ') \{ deptname[j] = s[i]; i++; j++; \}
|  |        deptname[j] = '\0';
|  |        i += 1; j =0;
|  |
|  |        WHILE(s[i] != '\0') \{ password[j] = s[i]; i++; j++; \}
|  |        password[j] = '\0';
|  |
|  |        IF (s[k] == 'd') form_dept(name,deptname,password);
|  |        ELSE form_group(name,deptname,password);
|  └──\}
└──\}
 /*************************************************************************
 The function of this form_sys is to declare a new node in the hierarchy
 and see that approprite addresses are set up. The system administrator
 controls has the power of the superuser in Unix. It could delete user in the
 system, delete files and perform various system administration work.
 *************************************************************************/
 form_sys(n,p)
 CHAR *n,*p;
┌──\{
|   STRUCT heirarchy *newnode;
|   STRUCT logon *newlogon;
|
|  ┌──IF (h_start ==NULL) \{
|  |     newnode = (STRUCT heirarchy *)malloc(SIZEOF(heirarchy_entry));
|  |    ┌──IF (!newnode) \{ printf("out of memory in form system administrator \n");
```

```
|  |    |                 exit(0);
|  |    └───\}
|  |     strcpy(newnode->username,n);
|  |     newnode->deptname[0] = '$';
|  |     newnode->groupname[0] = '$';
|  |     strcpy(newnode->key,"00");
|  |     strcpy(newnode->large,"01");
|  |     newnode->next = newnode->down = NULL;
|  |     newnode->head = NULL;
|  |     h_start = newnode;
|  |
|  |     newlogon = (STRUCT logon *)malloc(SIZEOF(logon_entry));
|  |    ┌──IF (!newlogon) \{
|  |    |                 printf("out of memory in form system for newlogon \n");
|  |    |                 exit(0);
|  |    └───\}
|  |     strcpy(newlogon->username,n);
|  |     strcpy(newlogon->password,p);
|  |     newlogon->heirarchy_ptr = newnode; /* the logon ptr points to new hierarchy node */
|  |     logon_start = logon_last = newlogon;
|  └───\}
|  ┌──ELSE \{ printf("check why is there another system administrator \n");
|  |        exit(0);
|  └──────────\}
└──────\}
  /*************************************************************************
  This form department function is called by the separate string and its
  primary function is to form the department head and perform various
  addresses set up in the hierarchical tree for the users. It declares a new
  node and copy the necessary information to identify the node and link it
  to the hierarchical tree.
  *************************************************************************/
  form_dept(n,d,p)
  CHAR *n,*d,*p;
┌────\{
|    STRUCT logon *newlogon, *curlogon;
|    STRUCT heirarchy *newnode, *curnode;
|
|    newnode = (STRUCT heirarchy *)malloc(SIZEOF(heirarchy_entry));
|  ┌──IF (!newnode) \{ printf("out of memory in form department\n ");
|  |                exit(0);
|  └───\}
|    strcpy(newnode->username,n);
|    strcpy(newnode->deptname,d);
|    newnode->groupname[0] = '$';
|    strcpy(newnode->key,"00");
|    strcpy(newnode->large,"01");
|    newnode->next = newnode->down = NULL;
|    newnode->head = NULL;
|
|    curnode =h_start;
|    IF (curnode->down == NULL)  curnode->down = newnode;
|  ┌──ELSE \{
```

```
|   |         curnode = curnode->down;
|   |         WHILE (curnode->next != NULL) curnode = curnode->next;
|   |         curnode->next = newnode;
|   └───\}
|    newlogon = (STRUCT logon *)malloc(SIZEOF(logon_entry));
|   ┌───IF (!newlogon) \{ printf(" out of memory in forming dept\n");
|   |       exit(0);
|   └───\}
|    strcpy(newlogon->username,n);
|    strcpy(newlogon->password,p);
|    newlogon->down = NULL;
|    newlogon->heirarchy_ptr = newnode;
|    IF (logon_start == NULL) logon_start = logon_last = newlogon;
|   ┌───ELSE \{
|   |         curlogon = logon_last;
|   |         curlogon->down = newlogon;
|   |         logon_last = newlogon;
|   └───\}
└───\}
  /****************************************************************************
  As the two functions described above, the form_group function is to
  declare a new node in the hierarchy and link them to the approprite position
  and it has the power of superuser on its subjects or group member under its
  hierarchy. But various users in other groups are not subjected to the control
  of this group leader.
  ****************************************************************************/
  form_group(n,d,p)
  CHAR *n,*d,*p;
┌───\{
|    STRUCT heirarchy *newnode,*curnode;
|    STRUCT logon *newlogon;
|
|    newnode = (STRUCT heirarchy *)malloc(SIZEOF(heirarchy_entry));
|   ┌───IF (!newnode) \{ printf("\n out of memory in forming group \n");
|   |                exit(0);
|   └───\}
|    strcpy(newnode->username,n);
|    strcpy(newnode->deptname,d);
|    strcpy(newnode->groupname,n);
|    strcpy(newnode->key,"00");
|    strcpy(newnode->large,"01");
|    newnode->next = newnode->down = NULL;
|    newnode->head = NULL;
|
|    curnode = h_start->down; /* on 1st dept */
|    WHILE( strcmp(curnode->deptname,d) != 0 && curnode->next != NULL)
|       curnode = curnode->next;
|
|    IF (curnode->down == NULL) curnode->down = newnode;
|   ┌───ELSE \{
|   |         curnode = curnode->down;
|   |         WHILE(curnode->next != NULL) curnode = curnode->next;
|   |         curnode->next = newnode;
```

```
|   └───\}
|
|    newlogon = (STRUCT logon *)malloc(SIZEOF(logon_entry));
|   ┌───IF (!newlogon) \{ printf(" out of memory in newlogon in form group \n");
|   |                exit(0);
|   └───\}
|    strcpy(newlogon->username,n);
|    strcpy(newlogon->password,p);
|    newlogon->heirarchy_ptr = newnode;
|    newlogon->down = NULL;
|    logon_last->down = newlogon;
|    logon_last = logon_last->down;
└───\}
  /*************************************************************************/
  form_member(n,d,g,p)
  CHAR *n,*d,*g,*p;
┌───\{
|    STRUCT heirarchy *newnode, *curnode;
|    STRUCT logon *newlogon;
|
|    newnode = (STRUCT heirarchy *)malloc(SIZEOF(heirarchy_entry));
|   ┌───IF (!newnode) \{ printf("out of memory in form member of newnode \n");
|   |                exit(0);
|   └───\}
|    strcpy(newnode->username,n);
|    strcpy(newnode->deptname,d);
|    strcpy(newnode->groupname,g);
|    strcpy(newnode->key,"00");
|    strcpy(newnode->large,"01");
|    newnode->next = newnode->down = NULL;
|    newnode->head = NULL;
|
|    curnode = h_start->down;
|    WHILE (strcmp(curnode->deptname,d) != 0 && curnode->next != NULL)
|        curnode = curnode->next;   /* find the deptname */
|
|    curnode = curnode->down; /* found the dept and search down for group */
|    WHILE (strcmp(curnode->username,g) != 0 && curnode->next != NULL)
|         curnode = curnode->next;
|    IF (curnode->down == NULL)   curnode->down = newnode;
|   ┌───ELSE \{
|   |       curnode = curnode->down;
|   |       WHILE (curnode->next != NULL)  curnode = curnode->next;
|   |       curnode->next = newnode;
|   └───\}
|    newlogon = (STRUCT logon *)malloc(SIZEOF(logon_entry));
|   ┌───IF (!newlogon) \{ printf("out of memory in newlogn of form member \n");
|   |                exit(0);
|   └───\}
|    strcpy(newlogon->username,n);
|    strcpy(newlogon->password,p);
|    newlogon->heirarchy_ptr = newnode;
|    newlogon->down = newlogon->down;
```

```
|     logon_last->down = newlogon;
|     logon_last = logon_last->down;
└────\}
  /**************************************************************************
  This batch process is called by the main function and it will separate
  the string send into system recognizable form so that the various command
  could be performed. It will simulate the eight file manipulation commands
  discussed in the thesis. There are
  1. Read a file  i.e      user r  filename
  2. Write a file          user w  filename
  3. Execute a file        user e  filename
  4. Create a file         user cr filename
  5. Copy a file           user cp sourcefilename targetfilename
  6. Delete a file         user d  filename
  7. List members          user lm
  8. List files            user lf
  9. Allow access for a file for individual member.
                           user ai targetuser filename access right.
     i.e user A allows user B to read his file name F1.
         A ai B F1 r
  10. Allow group access:
      This command allows the entire department or group to access his file
      command is : user ag targetgroup filename access right.
      i.e. user root department Comp to read and execute library file F2.
      command : root ag Comp F2 r
  11. The command to create a user in the system.
      name departmentname password.
  12. Change Directory : This command is designed for the user in the higher
                         hierarchy. It allows user in the higher hierarchy
                         to go to a directory that belongs to his subject.
      i.e. username1 cd username2
           In this case, user1 is the superior node of user2, thus, user1
           could change directory to user2 directory.
  **************************************************************************/
  batch_process()
┌────\{
|    CHAR s[80],name[MAXLEN],filename1[MAXLEN],filename2[MAXLEN],
|         loc[MAXLEN],ar;
|    REGISTER INT i,j;
|    i =1; j=0;
|    strcpy(s,str);
|
|    WHILE (s[i] != ' ') \{ name[j] = s[i]; i++; j++; \}
|    name[j] = '\0';
|
|    strcpy(loc,name);
|   ┌────IF ((fch = fopen(loc,"a")) == NULL) \{
|   |      printf("can't open file %s\n",loc);
|   |      exit(0);
|   └────\}
|    strcat(loc,"tree.dat");
|   ┌────IF ((f1 = fopen(loc,"a")) == NULL) \{
|   |      printf("can't open file %s\n",loc);
```

```
|  |      exit(0);
|  └───\}
|   first = s[i+1];
|   second = s[i+2];
|   i += 4; j =0;
|  ┌──WHILE (*(s+i) != ' ') \{
|  |     ┌──IF (isalnum(*(s+i))) \{ *(filename1+j) = *(s+i);
|  |     |                         i++; j++;
|  |     └──\}
|  |     ELSE i++;
|  └───\}
|   *(filename1+j) = '\0';
|  ┌──SWITCH(first) \{
|  |        ┌──CASE 'c': SWITCH(second) \{
|  |        |              CASE 'r':
|  |        |                     create_file(name,filename1);
|  |        |                     BREAK;
|  |        |              CASE 'p': i += 1; j=0;
|  |        |                     ┌──WHILE( *(s+i) != '\0') \{
|  |        |                     |         ┌──IF (isalnum(*(s+i))) \{
|  |        |                     |         |      *(filename2+j) = *(s+i);
|  |        |                     |         |      i++; j++;
|  |        |                     |         └──\}
|  |        |                     |        ELSE i++;
|  |        |                     └──\}
|  |        |               *(filename2+j) = '\0';
|  |        |                     copy_file(name,filename1,filename2);
|  |        |                     BREAK;
|  |        |              CASE 'd': change_dir(name,filename1);
|  |        |                     BREAK;
|  |        |              DEFAULT : fprintf(fch,"problem in inner switch ");
|  |        |                     printf("\n problem in inner switch ");
|  |        |                     exit(0);
|  |        └──────────────\}
|  |              BREAK;
|  |       CASE 'r': execute_file(name,filename1, 2);
|  |              BREAK;
|  |       CASE 'w': execute_file(name,filename1,3);
|  |              BREAK;
|  |       CASE 'e': execute_file(name,filename1,1);
|  |              BREAK;
|  |       CASE 'l': IF (second == 'f') list_file(name);
|  |              ELSE list_member(name);
|  |              BREAK;
|  |       CASE 'd': printf("will delete file %s by user %s\n",name,filename1);
|  |              delete_file(name,filename1);
|  |              BREAK;
|  |       CASE 'a': i += 1; j = 0;
|  |              WHILE (s[i] != ' ') \{ filename2[j] = s[i]; i++; j++; \}
|  |              filename2[j] = '\0'; ar = s[i+1];
|  |              allow_access(name,second,filename1,filename2,ar);
|  |              BREAK;
|  |       DEFAULT : fprintf(fch,"problem in outer switch of batch process \n");
```

```
|  |                    printf("problem in outer switch of batch process \n");
|  |                    exit(0);
|  └────\}
|    fclose(fch);
|    fclose(fl);
└────\}
  /***************************************************************************
  This function, upon receiving the separate string will check for the a
  approprite password in the system and call insertion to insert this file into
  the global binary file. Then it will call calkey to calculate the key of this
  new user and then call insertion again to insert the file into the directory
  of the user.
  ***************************************************************************/
  create_file(n,f)
  CHAR *n,*f;
┌────\{
|     STRUCT logon *curlogon;
|     STRUCT heirarchy *curnode, *heipt;
|     CHAR *int2bin();
|
|     curlogon = logon_start;
|     WHILE(strcmp(curlogon->username,n) != 0 && curlogon->down != NULL)
|       curlogon = curlogon->down;
|
|     curnode = heipt = curlogon->heirarchy_ptr;
|     insertion(f, &globalhead, heipt, 1); /* passed in for global bintree */
|     calkey(curnode,4,0, 0);
|     insertion(f, &(curnode->head), heipt, 0); /* insert in local bintree */
|     primeindex++;
|     RETURN;
└──────\}
  /****************************************************************************
  This calkey will receive the usernode from the calling function. It will
  calculate the the key based on the Chinese Remainder Theorem and use the file
  (unique) numbers from the file to calculate the Dj or the summation of all
  files in the directory. It started off by calculating L, the product of all
  file numbers and stored L in the string provided by usernode. Then using the
  old key and the file numbers in the directory, it will calculate the access
  rights of various files and stored them in the array of string. The Dj value
  is also calculated at the same time using Dj = L/Lj with L is the product
  of all Lj stored them into the array of string. The modulus of Dj, dj is also
  calculated using dj = Dj mod Lj and stored into the dj array. The xj is then
  calculated using the Eucledian algorithm and stored in the xj array. Thus, the
  key could be then calculated using key = Dj.xj.aij + Dk.xk.aik + ... mod L
  where j,k,l,m...........<= number of files in the directory.
  ***************************************************************************/
  calkey(cn, accright, givenfilenum, fromgroupaccess)
  STRUCT heirarchy *cn;
  INT accright, givenfilenum;
┌────\{
|    CHAR  *result, fn[MAX], sum[MAX], *smalldj, tempsum[MAX],
|          temp[MAX],temp1[MAX],*mul(),*int2bin(),*bdiv(),*add();
|    UNSIGNED LONG INT locdj,loclj,bin2int();
```

```
|    REGISTER INT i,k;
|    VOID calacc();
|
| ┌─IF (cn->head == NULL) \{
| |        ┌──IF (fromgroupaccess == 1) \{
| |        |     ┌──SWITCH(accright)\{
| |        |     |       CASE 1 : strcpy(cn->key,"0001");
| |        |     |       BREAK;
| |        |     |       CASE 2 : strcpy(cn->key,"0010");
| |        |     |       BREAK;
| |        |     |       CASE 3 : strcpy(cn->key,"0011");
| |        |     |       BREAK;
| |        |     |       DEFAULT: printf("error in calkey calculating fromgroupaccess\n");
| |        |     └───\}
| |        |     result = int2bin(givenfilenum);
| |        └───\}
| |        ┌──ELSE \{
| |        |      strcpy(cn->key,"0100");
| |        |      result = int2bin(prime[primeindex]);
| |        └───\}
| |       i = 0;
| |        ┌──WHILE (*(result+i) != '\0') \{
| |        |       cn->large[i] = *(result+i);
| |        |       i++;
| |        └───\}
| |       cn->large[i] = '\0';
| |       RETURN;
| └─────\}
|    num =0;
|  ┌──IF (accright != 0) \{
|  |
|  |    ┌──IF (fromgroupaccess == FALSE) \{
|  |    |      result = int2bin(prime[primeindex]);  /** convert the new filenum to bin **/
|  |    |      i =0;
|  |    |      ┌──WHILE (*(result+i) != '\0') \{
|  |    |      |       fn[i] = *(result+i);
|  |    |      |       i++;
|  |    |      └───\}
|  |    |      fn[i] = '\0';
|  |    └───\}
|  |    ┌──ELSE \{
|  |    |        result = int2bin(givenfilenum);
|  |    |        i = 0;
|  |    |          ┌──WHILE (*(result+i) != '\0') \{
|  |    |          |       fn[i] = *(result+i);
|  |    |          |       i++;
|  |    |          └───\}
|  |    |        fn[i] = '\0';
|  |    └───\}
|  |     strcpy(temp,cn->large);
|  |     strcpy(temp1,fn);
|  |     result = mul(temp1,temp);
|  |     /* result = mul(fn, cn->large) cal the sigma L **/
```

```
|  |     i= 0;
|  |    ┌──WHILE (*(result+i) != '\0') \{
|  |    |        cn->large[i] = *(result+i);
|  |    |        i++;
|  |    └──\}
|  |     cn->large[i] = '\0';
|  |     strcpy(globallarge,cn->large);
|  |     strcpy(globalkey,cn->key);
|  └──\}
|  ┌──ELSE \{
|  |     strcpy(globalkey,cn->key);
|  |     strcpy(globallarge,cn->large);
|  └──\}
|   calacc(cn->head);
|  ┌──IF (accright != 0) \{
|  |     IF (accright == 4)       strcpy(aij[num],"0100");
|  |     ELSE IF (accright == 3)  strcpy(aij[num]," 0011");
|  |     ELSE IF (accright == 2)  strcpy(aij[num],"0010");
|  |     ELSE IF (accright == 1)  strcpy(aij[num],"0001");
|  |     strcpy(temp,cn->large);
|  |     strcpy(temp1,fn);
|  |     result = bdiv(temp1,temp,1);
|  |     i = 0;
|  |    ┌──WHILE (*(result+i) != '\0') \{
|  |    |    dj[num][i] = *(result+i);
|  |    |    i++;
|  |    └──\}
|  |     dj[num][i] = '\0';
|  |     strcpy(temp,dj[num]);
|  |     smalldj = bdiv(temp1,temp,0);
|  |     i = 0;
|  |    ┌──WHILE( *(smalldj+i) != '\0') \{
|  |    |       temp[i] = *(smalldj+i);
|  |    |       i++;
|  |    └──\}
|  |     temp[i] ='\0';
|  |     locdj = bin2int(temp);
|  |     strcpy(temp,fn);
|  |     loclj = bin2int(temp);
|  |     result = int2bin(gcd(locdj,loclj)); /* cal xj and put to last array */
|  |     i=0;
|  |    ┌──WHILE ( *(result+i) != '\0') \{
|  |    |        xj[num][i] = *(result+i);
|  |    |        i++;
|  |    └──\}
|  |     xj[num][i] = '\0';
|  |     num++;
|  └───\}
|   /******* cal key now  *********/
|   strcpy(sum,"00");
|  ┌──FOR (i=0; i<num; i++) \{
|  |      strcpy(temp,dj[i]);
|  |      strcpy(temp1,aij[i]);
```

```
|   |      result = mul(temp,temp1);      /** dj[i] multiply aij[i] **/
|   |      k = 0;
|   |      ┌──WHILE ( *(result+k) != '\0') \{
|   |      |    *(temp+k) = *(result+k) ;
|   |      |    k++;
|   |      └──\}
|   |      *(temp+k) = '\0';
|   |                            /*  tempsum = mul(result, xj[i]); */
|   |      strcpy(temp1,xj[i]);
|   |      result = mul(temp,temp1);
|   |      k=0;
|   |      ┌──WHILE ( *(result+k) != '\0') \{
|   |      |    *(tempsum+k) = *(result+k);
|   |      |    k++;
|   |      └──\}
|   |      *(tempsum+k) = '\0';
|   |   ┌──IF (strcmp(tempsum, cn->large) > 0)  \{
|   |   |      strcpy(temp,cn->large);
|   |   |      strcpy(temp1,tempsum);
|   |   |                          /*tempsum = bdiv(cn->large, tempsum, 0); */
|   |   |      result = bdiv(temp,temp1,0);
|   |   |      k=0;
|   |   |       ┌──WHILE ( *(result+k) != '\0') \{
|   |   |       |       *(tempsum+k) = *(result+k);
|   |   |       |       k++;
|   |   |       └──\}
|   |   |      *(tempsum+k) = '\0';
|   |   |    /* find the modulus */
|   |   └──────\}
|   |      result = add(sum, tempsum);
|   |      k=0;
|   |      ┌──WHILE ( *(result+k) != '\0') \{
|   |      |      *(sum+k) = *(result+k);
|   |      |      k++;
|   |      └──\}
|   |      *(sum+k) = '\0';
|   └──\}
|    strcpy(temp,cn->large);
|    strcpy(temp1,sum);
|    result = bdiv(temp,temp1,0);
|    k = 0;
|   ┌──WHILE ( *(result +k) != '\0') \{
|   |    *(sum+k) = *(result+k);
|   |    k++;
|   └──\}
|    *(sum+k) = '\0';
|    strcpy(cn->key,sum); /* new key is found */
|    RETURN;
└──\}
 /****************************************************************************
 The use of this gcd is to calculate xj when it is called where
 djxj = 1 mod Lj. This function will then return the value of xj into the
 calling function.
```

```
 ************************************************************************/
 INT gcd(d,l)
 UNSIGNED LONG INT d,l;
┌──\{
|  UNSIGNED LONG INT x;
| ┌──FOR (x = 1; x <=l; x++) \{
| |      IF (((d*x) % l) == 1) RETURN(x);
| └──\}
|  fprintf(fch,"error in gcd with Dj = %ld and Lj = %ld\n",d,l);
|  printf("error in gcd with dj == %ld and lj == %ld \n",d,l);
|  exit(0);
└───\}
 /************************************************************************
 This function calacc will calculate the access rights of the various
 files in the directory. It receives the head node of the directory and
 using recursive technique to calculate the access rights.
 ************************************************************************/
 VOID calacc(head)
 STRUCT tree_node_rec *head;
┌──\{
|  CHAR fn[MAX], temp[MAX], *result,*int2bin(), *bdiv(), *smalldj,
|       temp1[MAX],temp2[MAX];
|  UNSIGNED LONG INT locdj,loclj;
|  REGISTER INT i;
|
|  IF (!head) RETURN;
|  result = int2bin(head->fnum);
|  i =0;
| ┌──WHILE ( *(result+i) != '\0') \{
| |    *(fn+i) = *(result+i);
| |    i++;
| └──\}
|  *(fn+i) = '\0';
|  strcpy(temp2,globalkey);
|  strcpy(temp1,fn);
|  result = bdiv(temp1,temp2,0);
|  /** aij[num] = bdiv(fn,globalkey, 0); cal big dj  = L div Lj ***/
|  i = 0;
| ┌──WHILE ( *(result+i) != '\0') \{
| |     aij[num][i] = *(result+i);
| |     i++;
| └──\}
|  aij[num][i] = '\0';
|  strcpy(temp1,fn);
|  strcpy(temp2,globallarge);
|  result = bdiv(temp1,temp2, 1);
|  /*dj[num] = bdiv(fn, globallarge, 1) cal small dj = dj div filenum */
|  i=0;
| ┌──WHILE ( *(result+i) != '\0') \{
| |      dj[num][i] = *(result+i);
| |      i++;
| └──\}
|  dj[num][i] = '\0';
```

```
|    strcpy(temp1,fn);
|    strcpy(temp2,dj[num]);
|                /*smalldj = bdiv(fn, dj[num], 0);*/
|    smalldj = bdiv(temp1,temp2);
|    i = 0;
|   ┌──WHILE ( *(smalldj+i) != '\0') \{
|   |    *(temp+i) = *(smalldj+i);
|   |    i++;
|   └──\}
|    *(temp+i) = '\0';
|    locdj = bin2int(temp); /* convert small dj to unsigned long int */
└──\}
    strcpy(temp,fn);
    loclj = bin2int(temp);
    result = int2bin(gcd(locdj,loclj));
    i = 0;
   ┌──WHILE ( *(result+i) != '\0') \{
   |    xj[num][i] = *(result+i);
   |    i++;
   └──\}
    xj[num][i] = '\0';
     /* xj[num] = int2bin(gcd(locdj, head->fnum)) cal xj and store in the array */
    num++;
    calacc(head->lpt);
    calacc(head->rpt);
    RETURN;
  \}
  /************************************************************************
  This execute_file will carry out the request by the user and perform the
  execute function. It first check the user's password for validity of the
  command. Then it will find the usernode in the hierarchy structure. If this
  file is found in his own directory, then he can access it. Else it will
  go to the global binary file directory to check for this file and retrieve
  the address that points to the owner of this file. Comparison is made on the
  user and the owner of this file. If the owner of this file is the subject of
  this user, then user has exclusive access rights on this file. Else, this
  the user request is rejected.
  ************************************************************************/
  execute_file(n,f, accright)
  CHAR *n,*f;
  INT accright;
┌──\{
|    STRUCT logon *curlogon;
|    STRUCT heirarchy *curnode;
|    STRUCT tree_node_rec *accessnode, *stack[MAXSTACK], *loc, *find_node();
|    CHAR *result,temp[MAX],temp1[MAX],*bdiv(), *int2bin();
|    INT i, found, stack_top;
|    REGISTER INT k;
|
|    curlogon = logon_start;
|    WHILE ( strcmp(curlogon->username,n) != 0 && curlogon->down != NULL)
|            curlogon = curlogon->down;
|    curnode = curlogon->heirarchy_ptr;
```

```
|    accessnode = curnode->head;
|  ┌──IF (accessnode == NULL) \{
|  |     fprintf(fch,"%s has no files in the directory \n",n);
|  |     RETURN;
|  └──\}
|    loc = find_node(accessnode,f, &found, stack, &stack_top, 1);
|  ┌──IF (found == TRUE) \{
|  |     result = int2bin(loc->fnum);  /* convert filenum to string */
|  |     k = 0;
|  |   ┌──WHILE ( *(result+k) != '\0' ) \{
|  |   |     *(temp+k) = *(result+k);
|  |   |     k++;
|  |   └──\}
|  |     *(temp+k) = '\0';
|  |     strcpy(temp1,curnode->key);
|  |     result = bdiv(temp,temp1,0);
|  |     /* bdiv(result, curnode->key, 0) cal the acc right */
|  |     k=0;
|  |   ┌──WHILE ( *(result+k) != '\0') \{
|  |   |     *(temp+k) = *(result+k);
|  |   |     k++;
|  |   └──\}
|  |     *(temp+k) = '\0';
|  |     i = bin2int(temp); /* convert acc right to int */
|  |   ┌──IF ( i >= accright) \{
|  |   |     IF (accright == 1) strcpy(temp,"execute");
|  |   |     ELSE IF (accright == 2) strcpy(temp,"read");
|  |   |     ELSE IF (accright ==3) strcpy(temp,"write");
|  |   |     printf("\nfile %s is allowed %s by user %s \n",f,temp,n);
|  |   |     RETURN;
|  |   └───\}
|  |    ┌──ELSE \{ printf("user %s is not allowed execute on file %s ",n,f);
|  |    |        RETURN;
|  |    └──\}
|  └──\}
|  ┌──ELSE \{
|  |       loc = find_node(globalhead, f, &found, stack, &stack_top, 1);
|  |        ┌──IF (found == TRUE) \{
|  |        |   ┌──IF ((curnode->deptname[0] == '$') && (curnode->groupname[0] == '$')) \{
|  |        |   |        printf("user %s is allowed to access file %s with rights %d \n",n,f,accright)

|  |        |   |        RETURN;
|  |        |   └──\}
|  |        |   ┌──ELSE IF( (curnode->groupname[0] == '$') && (strcmp(curnode->deptname, loc->o

|  |        |   |  deptname) == 0)) \{
|  |        |   |        printf("user %s is allowed to access file %s with rights %d \n", n, f, accri

|  |        |   |        RETURN;
|  |        |   └──\}
|  |        |   ┌──ELSE IF ( strcmp(curnode->username,loc->ownerpt->groupname) == 0) \{
|  |        |   |        printf("user %s is allowed to access file %s with rights %d \n", n, f, accri
```

```
|  |           |         RETURN;
|  |           └───\}
|  └───\}
└───\}
   /**********************************************************************
   This copy file function is called by the batch_process and its main
   function is to copy the file1 to file2. After being invoked, it will
   search the list of all users in the system, equivalent the etc/passwd
   in the Unix system, after verifying the users and the password, the
   function will be using the names of file1 to find the file in the local
   directory, if the file is found, then it will create another node in the
   local directory and call crete file function to create a node in the
   directory and perform key calculation by calling the calkey and insertion
   to insert the file in the local directory.
  **********************************************************************/
  copy_file(n,f1,f2)
  CHAR *n,*f1,*f2;
┌───\{
|   STRUCT logon *curlogon;
|   STRUCT heirarchy *curnode;
|   STRUCT tree_node_rec *loc,*find_node(),*stack[MAXSTACK];
|   INT found, stack_top;
|
|   curlogon = logon_start;
|   WHILE( strcmp(curlogon->username,n) != 0 && curlogon->down != NULL)
|      curlogon = curlogon->down;
|
|   curnode = curlogon->heirarchy_ptr;
|  ┌───IF (curnode->head == NULL) \{
|  |     printf("No file in the dir of %s \n",n);
|  |     RETURN;
|  └───\}
|   loc = find_node(curnode->head, f1, &found, stack, &stack_top, 1);
|   printf("node copied is %s\n",loc->info);
|   exit(0);
└───\}
  ┌───IF (found==TRUE) \{
  |     create_file(n,f2);
  |     RETURN;
  └───\}
  ┌───ELSE \{
  |      printf(" file %s not found in the directory \n",f1);
  |      RETURN;
  └───\}
  \}
    /**********************************************************************
  This list file function is to list the file that the user has in his
  directory. It will list the file names that are accessible by the user
  as well as listing the access rights of the user towards that file. It did
  this by retrieving the key of the user, and retrieving the file number of the
  file in the directory and perform a calculation of

  access right = key mod filenumber.
```

```

Then this listing is listed on the file.
*****************************************************************************/
list_file(n)
CHAR *n;
┌──\{
│   STRUCT logon *curlogon;
│   STRUCT heirarchy *curnode;
│
│   curlogon = logon_start;
│   WHILE( strcmp(curlogon->username,n) != 0 && curlogon->down != NULL)
│      curlogon = curlogon->down;
│
│   curnode = curlogon->heirarchy_ptr;
│  ┌──IF (curnode->head == NULL) \{
│  │     printf("no files in the directory of user %s \n", n);
│  │     RETURN;
│  └──\}
│   printtree(curnode->head,curnode->key);
│   RETURN;
└──\}
/*****************************************************************************
This printtree is called by the list file function. If there are more file
in the binary local directory tree, then it will call itself recursively to
print more files names and access rights.
*****************************************************************************/
printtree(head, key)
STRUCT tree_node_rec *head;
CHAR *key;
┌──\{
│    CHAR localkey[MAX], *result, *bdiv(), temp[MAX];
│    REGISTER INT i;
│
│    IF (!head) RETURN;
│
│    result = int2bin(head->fnum);
│    i = 0;
│   ┌──WHILE ( *(result+i) != '\0') \{
│   │     *(temp+i) = *(result+i);
│   │     i++;
│   └──\}
│    *(temp+i) = '\0';
│    strcpy(localkey,key);
│    result = bdiv(temp,localkey,0);
│    /**result = bdiv(result, key, 0) */
│    fprintf(fch,"File name -> %s and access right is %s\n", head->info,result);
│    strcpy(localkey,key);
│    printtree(head->lpt,localkey);
│    printtree(head->rpt,localkey);
└──\}
/*****************************************************************************
This list member function is called by the batch_process function, it
receives information on the name of the user. The function will then
```

```
search for the user in the logon file and verify the password. If it is
correct, the function will then follow the addresses of the hierarchy and
print the names of the user and their department.
*************************************************************************/
list_member(n)
CHAR *n;
┌──\{
|    STRUCT logon *curlogon;
|    STRUCT heirarchy *curnode;
|
|    curlogon = logon_start;
|    WHILE( strcmp(curlogon->username,n) != 0 && curlogon->down != NULL)
|       curlogon = curlogon->down;
|
|    curnode = curlogon->heirarchy_ptr;
|   ┌──IF (curnode->down == NULL) \{ printf(" no members in group %s \n",n);
|   |     RETURN;
|   └──\}
|   ┌──IF (curnode->groupname[0] == '$' && curnode->deptname[0] == '$') \{
|   |    fprintf(fch,"listing all members of system \n");
|   |    printing(curnode);
|   |    RETURN;
|   └──\}
|   ┌──ELSE IF (curnode->groupname[0] == '$') \{
|   |          fprintf(fch,"listing all members of dept %s \n",curnode->deptname);
|   |          printing(curnode);
|   |          RETURN;
|   └──\}
|   ┌──ELSE \{         /***** curnode->groupname[0] != '$' ******/
|   |     fprintf(fch,"listing all members of group %s \n",curnode->groupname);
|   |     curnode = curnode->down;
|   |    ┌──WHILE(curnode != NULL) \{ printf(" %s \n",curnode->username);
|   |    |      curnode = curnode->next;
|   |    └──\}
|   |     RETURN;
|   └──\}
└──\}
/************************************************************************
This change directory function is called by the batch process function and
receive names of the superios node and name of the inferior node. If this
relationship holds, the command would be obeyed.
*************************************************************************/
change_dir(n,n1)
CHAR *n,*n1;
┌──\{
|   STRUCT heirarchy *user1, *user2;
|   STRUCT logon *curlogon;
|
|   curlogon = logon_start;
|   WHILE ( strcmp(curlogon->username,n) != 0 && curlogon->down != NULL)
|        curlogon = curlogon->down;
|   user1 = curlogon->heirarchy_ptr;
|
```

```
|    curlogon = logon_start;
|    WHILE( strcmp(curlogon->username,n1) != 0 && curlogon->down != NULL)
|           curlogon  = curlogon->down;
|    user2 = curlogon->heirarchy_ptr;
|
|   ┌──IF (user1->deptname[0] == '$' && user1->groupname[0] == '$') \{
|   |      fprintf(fch,"allow change dir \n");
|   |      user1 = user2; RETURN;
|   └──\}
|   ┌──ELSE IF (user1->groupname[0] == '$' && user2->groupname[0] == '$') \{
|   |       fprintf(fch,"change dir not allowed \n");
|   |      RETURN;
|   └──\}
|    ELSE IF (user1->groupname[0] == '$' && user2->groupname[0] != '$' &&
|              ┌──strcmp(user1->deptname,user2->deptname) == 0) \{
|              |  fprintf(fch,"allow change dir \n");
|              |  RETURN;
|    \}
|   ┌──ELSE IF ( strcmp(user1->username,user2->groupname) == 0) \{
|   |          fprintf(fch,"allowed accessed \n");
|   |          RETURN;
|   └──\}
|   ┌──ELSE \{ fprintf(fch," no such cases between %s and %s \n",n,n1);
|   |        RETURN;
|   └──\}
└──\} /********* end of function ****************/
   /*********************************************************************
   This delete_file function is called by the batch_process and will
   check for the user in the logon list to ensure security. Then it will
   search the file in the global directory. If the global directory contains
   the file, then this file will be deleted. Any member that has this file will
   have their file deleted and their keys would be recalculated accordingly.
   It also performs necessary checking on the validity of the user and whether
   the file is owned by the user. If validity test fails, then the delete
   request is not honored.
   *********************************************************************/
   delete_file(n,f)
   CHAR *n,*f;
┌──\{
|    STRUCT logon *curlogon;
|    STRUCT heirarchy *curnode, *cn;
|    STRUCT tree_node_rec *loc, *stack[MAXSTACK];
|    CHAR *result,*lock,*bdiv(),templock[MAX],temp1[MAX],temp2[MAX];
|    INT stack_top, found, i;
|    REGISTER INT k;
|
|    curlogon = logon_start;
|    WHILE (strcmp(curlogon->username,n) != 0 && curlogon->down != NULL)
|          curlogon = curlogon->down;
|
|    curnode = curlogon->heirarchy_ptr;
|
|    loc = find_node(globalhead, f, &found, stack, &stack_top, 1);
```

```
|  ┌──IF ((found == TRUE) && (loc->ownerpt == curnode)) \{
|  |     printf("user %s owns file %s and deleting..... \n",n,f);
|  |
|  |     loc = find_node(curnode->head, f, &found, stack, &stack_top, 1);
|  |      /* find file in local bin tree */
|  |
|  |     lock = int2bin(loc->fnum); /* convert fnum to bin */
|  |     k = 0;
|  |     ┌──WHILE ( *(lock+k) != '\0') \{
|  |     |     *(templock+k) = *(lock+k);
|  |     |     k++;
|  |     └──\}
|  |     *(templock+k) = '\0';
|  |     deletion(f, &(globalhead),1); /* deleting the global file */
|  |     curlogon = logon_start;
|  |     ┌──WHILE(curlogon != NULL) \{
|  |     |     cn = curlogon->heirarchy_ptr;
|  |     |     strcpy(temp1,templock);
|  |     |     strcpy(temp2,cn->large);
|  |     |     result = bdiv(temp1,temp2,0);
|  |     |     /*result = bdiv(lock, cn->large, 0)*/
|  |     |     i=0; found = FALSE;
|  |     |    ┌──WHILE (*(result+i) != '\0') \{
|  |     |    |     IF (*(result+i) == '1') found = TRUE;
|  |     |    |     IF (found == TRUE) BREAK; /* to check whether divisible by fnum */
|  |     |    |     i++;
|  |     |    └──\}
|  |     |    ┌──IF (found == FALSE) \{
|  |     |    |     deletion(f, &(cn->head),0); /* remainder == 0 */
|  |     |    |     strcpy(temp1,templock);
|  |     |    |     strcpy(temp2, cn->large);
|  |     |    |     result = bdiv(temp1,temp2, 1);
|  |     |    |     k = 0;
|  |     |    |    ┌──WHILE ( *(result+k) != '\0') \{
|  |     |    |    |     *(temp1+k) = *(result+k);
|  |     |    |    |     k++;
|  |     |    |    └──\}
|  |     |    |     *(temp1+k) = '\0';
|  |     |    |     strcpy(cn->large,temp1);
|  |     |    |     /*curnode->large = bdiv(lock, curnode->large, 1)*/
|  |     |    |     IF (cn->head != NULL) calkey(cn, 0, 0, 0);
|  |     |    └──\}
|  |     |    curlogon = curlogon->down;
|  |     └──\}
|  └──\}
|  ┌──ELSE IF( (found == TRUE ) && (loc->ownerpt != curnode)) \{
|  |      loc = find_node(curnode->head, f, &found, stack, &stack_top, 1);
|  |      ┌──IF (found == TRUE) \{
|  |      |     deletion(f, &(curnode->head),0);
|  |      |     strcpy(temp1,templock);
|  |      |     strcpy(temp2,curnode->large);
|  |      |     result = bdiv(temp1,temp2,1);
|  |      |     k=0;
```

```
|   |         |         ┌───WHILE ( *(result+k) != '\0') \{
|   |         |         |      *(templ+k) = *(result+k);
|   |         |         |      k++;
|   |         |         └───\}
|   |         |         *(templ+k) = '\0';
|   |         |         strcpy(curnode->large,templ);
|   |         |         /*curnode->large = bdiv(lock, curnode->large, 1) */
|   |         |         IF (curnode->head != NULL ) calkey(curnode, 0, 0, 0);
|   |         └───\}
|   |         ┌───ELSE \{
|   |         |      printf("file not found in local bin dir of %s\n",n);
|   |         |      RETURN;
|   |         └───\}
|   └───\}
|   ┌───ELSE IF (found == FALSE) \{ /* can't find the file in global bin tree */
|   |        loc = find_node(curnode->head, f, &found, stack, &stack_top, 1);
|   |         ┌───IF (found == TRUE) \{
|   |         |      deletion(f, &(curnode->head),0);
|   |         |      printf("file %s deleted in local directory \n", f);
|   |         |      IF (curnode->head != NULL) calkey(curnode, 0, 0, 0);
|   |         |      RETURN;
|   |         └───\}
|   |         ┌───ELSE \{
|   |         |      printf("file does not exist in both local and global directory \n");
|   |         |      RETURN;
|   |         └───\}
|   └───\}
└───\}
  /*****************************************************************************
  This function is directly called by the batch_process function and its main
  function is to determine the requested user in the system and find the file
  in his directory. Only file that are present in the requested user's
  directory are allowed to proceed. It will also determine the group access
  or individual access this user requested and will call the approprite
  function to proceed with the processing of the access.
  *****************************************************************************/
  allow_access(n,gori,uname,fname,accessright)
  CHAR *n,gori,*uname,*fname,accessright;
┌───\{
|    STRUCT logon *curlogon;
|    STRUCT heirarchy *curnode, *usernode;
|    STRUCT tree_node_rec *ghead, *loc, *find_node(), *stack[MAXSTACK];
|    INT num, found, stack_top;
|
|    curlogon = logon_start;
|    WHILE( strcmp(curlogon->username,n) != 0 && curlogon->down != NULL)
|           curlogon = curlogon->down;
|
|    curnode = curlogon->heirarchy_ptr;
|   ┌───IF (curnode->head == NULL) \{
|   |      fprintf(fch,"no files in the dir of user %s in allow access \n",n);
|   |      RETURN;
|   └───\}
```

```
      ghead = globalhead;
      loc = find_node(ghead, fname, &found, stack, &stack_top, 1);
      IF (found == FALSE ) \{
          printf("File %s not found in allow access \n", fname);
          RETURN;
      \}
      IF ((found == TRUE) && (loc->ownerpt != curnode)) \{
          printf("file %s is found but not owned by user %s \n",fname, n);
          RETURN;
      \}
      loc = find_node(curnode->head, fname, &found, stack, &stack_top, 1);
      IF (found == FALSE) \{
          printf("file %s is not found in the local direc\n",fname);
          RETURN;
      \}
      curlogon = logon_start;
      WHILE (strcmp(curlogon->username,uname) != 0 && curlogon->down != NULL)
             curlogon = curlogon->down;
      IF (curlogon->down == NULL && strcmp(curlogon->username,uname) != 0 ) \{
          IF (gori == 'g') fprintf(fch,"no such group \n");
          ELSE fprintf(fch,"no such individual \n");
          RETURN;
      \}
      SWITCH(accessright) \{
          CASE 'e': num = 1;
                    BREAK;
          CASE 'r': num = 2;
                    BREAK;
          CASE 'w': num = 3;
                    BREAK;
      \}
      usernode = curlogon->heirarchy_ptr;
      IF (gori == 'g') groupaccess(usernode,num,loc->fnum,loc->info, 1);
      ELSE groupaccess(usernode,num,loc->fnum,loc->info, 0);
      RETURN;
  \}
  /*****************************************************************************
  This groupaccess is called by the allow access and it will determine the
  whether this is a group access or individual access. If individual access is
  requested, it will run once by calling insertion function to insert the file
  in the target user's directory and call calkey function to recalculate the
  key of the user again. If group access is encountered, it will keep calling
  groupaccess recursively to perform the above function.
  *****************************************************************************/
  groupaccess(root,givenright,or_fnumber,or_fname, grpacc)
  STRUCT heirarchy *root;
  INT givenright,or_fnumber, grpacc;
  CHAR *or_fname;
  \{
      STRUCT heirarchy *heipt;
      STRUCT tree_node_rec *loc, *stack[MAXSTACK],*find_node();
      INT stack_top, found;

```

```
|    ┌──IF (!root)\{
|    |            printf("inside root has nothing \n");
|    |            RETURN;
|    └──\}
|     heipt = root;
|    ┌──IF (root->head == NULL) \{
|    |     calkey(root,givenright,or_fnumber,1);
|    |     insertion(or_fname, &(root->head), heipt, 0);
|    |     loc = find_node(root->head,or_fname,&found,stack,&stack_top,1);
|    |     loc->fnum = or_fnumber;
|    |    ┌──IF (grpacc == TRUE) \{
|    |    |    groupaccess(root->next,givenright,or_fnumber,or_fname,1);
|    |    |    groupaccess(root->down,givenright,or_fnumber,or_fname,1);
|    |    └──\}
|    |     ELSE RETURN;
|    └──\}
|    ┌──ELSE \{
|    |       calkey(root, givenright, or_fnumber, 1);
|    |       insertion(or_fname, &(root->head), heipt, 0);
|    |       loc = find_node(root->head,or_fname,&found,stack,&stack_top,1);
|    |       loc->fnum = or_fnumber;
|    |    ┌──IF (grpacc == TRUE) \{
|    |    |    printf("inside the groupaccess of more than one file\n");
|    |    |    printf("user name is %s\n",root->username);
|    |    |    groupaccess(root->next, givenright, or_fnumber, or_fname,1);
|    |    |    groupaccess(root->down, givenright, or_fnumber, or_fname,1);
|    |    └───\}
|    └──\} /** else loop **/
└──\}
   /*************************************************************************
   This printing is called by the main program to print all users in the
   hierarchy for their name department name and group name. It will call itself
   recursively.
   *************************************************************************/
   printing(root)
   STRUCT heirarchy *root;
┌──\{
|   IF (!root) RETURN;
|   fprintf(fch,"The name is %s \n",root->username);
|   fprintf(fch,"The deptname is %s \n",root->deptname);
|   fprintf(fch,"The groupname is %s \n",root->groupname);
|   fprintf(fch,"****************** \n");
|   printing(root->down);
|   printing(root->next);
└──\}
   /*************************************************************************
   This print logon function is called by the main program and it will print
   out all the users name and password in the \dev\passwd directory. It is only
   supposed to be called by the system administrator.
   *************************************************************************/
   print_logon(root)
   STRUCT logon *root;
┌──\{
```

```
|   STRUCT logon *curlogon;
|   IF (!root) RETURN;
|   curlogon = root;
| ┌──DO \{
| |     fprintf(fch,"name is %s \n",curlogon->username);
| |     fprintf(fch,"password is %s \n",curlogon->password);
| |     fprintf(fch,"username is %s \n",curlogon->heirarchy_ptr->username);
| |     fprintf(fch,"deptname is %s \n",curlogon->heirarchy_ptr->deptname);
| |     fprintf(fch,"groupname is %s \n",curlogon->heirarchy_ptr->groupname);
| |     fprintf(fch,"key is %ld \n",curlogon->heirarchy_ptr->key);
| |     curlogon = curlogon->down;
| └──\} WHILE (curlogon != NULL);
└──\}
  /***************************************************************************
  This print function is called by various tree manipulation function in the
  program. It will print the name of the files in the local as well as global
  directory if call appropritely.
  ***************************************************************************/
  print (s,global)
  CHAR *s;
┌──\{  REGISTER INT i;
|     FILE *fout;
|
|     IF (global == TRUE) fout = fg;
|     ELSE fout = fl;
|     i =0;
|    ┌──WHILE (s[i] != '\0')\{
|    |    fprintf (fout, "%c", s[i]);
|    |    i +=1;
|    └──\}
|    fprintf(fout, "\n");
|    RETURN;
└──\}
  /***************************************************************************
  This find_node function is called by various tree manipulation function and
  return a file node record type once it is found. When this function is called
  the calling function will pass the name of the file, and the stack to store the
  pointer for the file. The head is the pointer of the head node in the tree,
  whether it is a global binary tree or local binary tree.
  ***************************************************************************/
  STRUCT tree_node_rec *find_node (head, info, found, stack, stack_top, ori)
  CHAR info[];
  INT *found;
  STRUCT tree_node_rec *stack[], *head;
  INT *stack_top;
  INT ori;
┌──\{ STRUCT tree_node_rec *pre, *cur;
|    STRUCT tree_node_rec *temp_stack[MAXSTACK];
|    INT i,temp_top,temp_found;
|
|    pre = cur = head;
|    temp_top = -1;
|    temp_found = FALSE;
```

```
|  ┌──WHILE ((temp_found != TRUE) && (cur != NULL))\{
|  |      temp_top++;
|  |      temp_stack[temp_top] = cur;
|  |      IF (strcmp(cur->info,info) == 0) temp_found = TRUE;
|  |      ┌──ELSE \{
|  |      |       pre = cur;
|  |      |       IF (strcmp(cur->info,info) < 0)  cur = cur->rpt;
|  |      |       ELSE  cur = cur->lpt;
|  |      └──\}
|  └──\} /* while loop */
|    *found = temp_found;
|    *stack_top = temp_top;
|    FOR (i=0; i<=temp_top; i++)  stack[i] = temp_stack[i];
|    IF ((temp_found == TRUE) && (ori == 1)) RETURN(cur);
|    ELSE RETURN(pre);
└──\} /* end of find_node */
  /**************************************************************************
  This insertion function is called by various file manipulation function. The
  parameter that passed in is the name of the file, s. The head of the tree and
  the pointer that points to the user node. For global file insertion, it will
  store the pointer in the global file node.
  **************************************************************************/
  insertion (s, head,heipt,globalbin)
  CHAR *s;
  STRUCT tree_node_rec **head;
  STRUCT heirarchy *heipt;
  INT globalbin;
┌──\{ STRUCT tree_node_rec *find_node(), *new_node, *loc, *stack[MAXSTACK];
|    INT critical, found, critical_node, stack_top;
|    FILE *fout;
|
|    IF (globalbin == TRUE) fout = fg;
|    ELSE fout = fl;
|    loc = find_node (*head, s, &found, stack, &stack_top, 0);
|    IF (found == TRUE) fprintf (fout, " is already existed. No insertion !\n\n");
|  ┌──ELSE \{
|  |      new_node = (STRUCT tree_node_rec *)malloc(SIZEOF(tree_node_type));
|  |      ┌──IF (!new_node)\{
|  |      |       fprintf(fout, "out of memory in insertion \n\n");
|  |      |       exit(0);
|  |      └──\}
|  |       strcpy (new_node->info, s);
|  |       new_node->lpt = NULL;
|  |       new_node->rpt = NULL;
|  |       new_node->tag = 0;
|  |       ┌──IF (globalbin == TRUE)  \{
|  |       |      new_node->ownerpt = heipt;
|  |       |      new_node->fnum = 0;
|  |       └──\}
|  |       ┌──ELSE \{
|  |       |       new_node->ownerpt = NULL;
|  |       |       new_node->fnum = prime[primeindex];
|  |       └──\}
```

```
|   |       IF (*head == NULL)  *head = new_node;
|   |       ┌──ELSE \{
|   |       |        IF (strcmp (loc->info, s) < 0) loc->rpt = new_node;
|   |       |        ELSE loc->lpt = new_node;
|   |       |        stack_top++;
|   |       |        stack[stack_top] = new_node;
|   |       |        modify_tag (*head, INS, &critical, stack, stack_top, &critical_node);
|   |       |        IF (critical == TRUE)
|   |       |        IF (globalbin == 1) balance_tree (head, INS, stack, critical_node,1);
|   |       |        ELSE balance_tree(head, INS, stack, critical_node,0);
|   |       └──\}
|   |       ┌──IF (globalbin == 1) \{
|   |       |     print_tree (0, *head,1);
|   |       |     fprintf (fout, "\n");
|   |       └──\}
|   |       ┌──ELSE \{
|   |       |       print_tree(0, *head, 0);
|   |       |       fprintf(fout, "\n");
|   |       └──\}
|   └───────\}
|    RETURN;
└──\}  /* end of insertion */
  /*************************************************************************
  This modify tag function is to modify the tag of the file in both the global
  and local file. The idea is that for a balance tree, on any node in the tree,
  the difference between the number of nodes on the right and the number of
  nodes on the left must not be greater than 1.
  *************************************************************************/
  modify_tag (head, process, critical, stack, stack_top, critical_node)
  INT process, stack_top, *critical, *critical_node;
  STRUCT tree_node_rec *stack[], *head;
┌──\{ INT pre, temp_top, temp_critical_node, stop, temp_critical;
|
|    pre = stack_top;
|    temp_top = stack_top-1;
|    temp_critical = FALSE;
|    stop = FALSE;
|    loopagain:           /* the famous loop starts here !!!! */
|      IF ((process == DEL) && (stack[temp_top])->tag == 0 )  stop = TRUE;
|     ┌──IF (strcmp(stack[temp_top]->info, stack[pre]->info) > 0) \{
|     |    IF (process == INS) (stack[temp_top])->tag--;
|     |    ELSE (stack[temp_top])->tag++;
|     └──\}
|     ┌──ELSE\{
|     |       IF (process == INS) (stack[temp_top])->tag++;
|     |       ELSE (stack[temp_top])->tag--;
|     └──\}
|     ┌──IF (abs(stack[temp_top]->tag) > 1 ) \{
|     |       temp_critical_node = temp_top;
|     |       temp_critical = TRUE;
|     └──\}
|      IF ((stop == TRUE) || (temp_critical == TRUE) || (stack[temp_top] == head)
|         || ((stack[temp_top]->tag == 0) && (process == INS)))
```

```
|        GOTO retval;
|      ┌──ELSE \{
|      |      pre = temp_top;
|      |      temp_top--;
|      |      GOTO loopagain;
|      └──\}
|      retval: *critical = temp_critical;
|              *critical_node = temp_critical_node;
|      RETURN;
└──\}  /*** end of modify_tag ***/
   /****************************************************************************
   The single left tree rotation function is one of the tree manipulation
   function that is called by balance tree. If the balance tree function
   determines that the tree is not balance, then it needs to be rotated.
   ****************************************************************************/
   single_left (head, stack, critical_node)
   STRUCT tree_node_rec *stack[], **head;
   INT    critical_node;
┌──\{
|    INT pivot;
|    STRUCT tree_node_rec *pivot_right;
|
|    pivot = critical_node + 1;
|    pivot_right = stack[pivot]->rpt;
|    stack[pivot]->rpt = stack[critical_node];
|    stack[critical_node]->lpt = pivot_right;
|    IF (stack[critical_node] == *head)   *head = stack[pivot];
|    ELSE IF (stack[critical_node - 1]->lpt == stack[critical_node])
|        stack[critical_node - 1]->lpt = stack[pivot];
|    ELSE stack[critical_node - 1]->rpt = stack[pivot];
|    /* end if */
|    stack[critical_node]->tag = 0;
|    stack[pivot]->tag = 0;
└──\} /* end of single_left */
   /****************************************************************************
   The single right rotation function will rotate the tree once it is out of
   balance. It will bring the parent node and put into the right child.
   ****************************************************************************/
   single_right (head, stack, critical_node)
     STRUCT tree_node_rec *stack[], **head;
     INT    critical_node;
┌──\{
|    INT pivot;
|    STRUCT tree_node_rec *pivot_left;
|
|    pivot = critical_node + 1;
|    pivot_left = stack[pivot]->lpt;
|    stack[pivot]->lpt = stack[critical_node];
|    stack[critical_node]->rpt = pivot_left;
|    IF (stack[critical_node] == *head) *head = stack[pivot];
|    ELSE IF (stack[critical_node - 1]->lpt == stack[critical_node])
|        stack[critical_node - 1]->lpt = stack[pivot];
|    ELSE stack[critical_node - 1]->rpt = stack[pivot];
```

```
   /* end if */
   stack[critical_node]->tag = 0;
   stack[pivot]->tag = 0;
\}  /* end of single_right */
/*****************************************************************************
The double left rotation will rotate once and then call the single left
rotation to continue rotating. The variable that sent in and out are the
stack of tree node pointers that point to the path of affected nodes.
*****************************************************************************/
double_left (head, stack, critical_node)
STRUCT tree_node_rec *stack[], **head;
INT    critical_node;
\{
   INT pivot, zeroed, i;
   STRUCT tree_node_rec *pivot_right;
   STRUCT tree_node_rec *loc_stack[MAXSTACK];

   pivot = critical_node + 1;
   pivot_right = stack[pivot]->rpt;
   FOR (i = 0; i < MAXSTACK; i++) loc_stack[i] = stack[i];
   IF (pivot == NULL)  zeroed = FALSE;
   ELSE IF ((pivot_right != NULL) && (pivot_right->tag == 1)) zeroed = TRUE;
   ELSE zeroed = FALSE;
   /* end if */
   stack[critical_node]->lpt = pivot_right;
   stack[pivot]->rpt = pivot_right->lpt;
   pivot_right->lpt  = stack[pivot];
   loc_stack[pivot]   = pivot_right;
   loc_stack[pivot+1] = stack[pivot];

   single_left (head, loc_stack, critical_node);

   IF ((stack[critical_node]->rpt != NULL) && (stack[critical_node]->lpt == NULL))
        stack[critical_node]->tag = 1;
   ELSE IF ((stack[critical_node]->rpt == NULL) && (stack[critical_node]->lpt != NULL))
        stack[critical_node]->tag = -1;
   ELSE stack[critical_node]->tag = 0;
   /* end if */
   IF ((stack[pivot]->lpt == NULL) && (stack[pivot]->rpt != NULL))
      stack[pivot]->tag = 1;
   ELSE IF ((stack[pivot]->tag == 1) && (stack[pivot]->lpt != NULL) &&
             (stack[pivot]->lpt->tag != 0))
        stack[pivot]->tag = -1;
   ELSE IF ((stack[pivot]->lpt != NULL) && (stack[pivot]->rpt == NULL))
        stack[pivot]->tag = - 1;
   ELSE IF ((stack[pivot]->tag == 1) && (zeroed == TRUE)
            && (stack[pivot]->lpt != NULL) && (stack[pivot]->lpt->tag == 0))
            stack[pivot]->tag = -1;
   ELSE stack[pivot]->tag = 0;
   /* end if */
\}  /* end of double_left */
/*****************************************************************************
This double right rotation will rotate once and then call single right
```

```
rotation to continue the second rotation. The stack of pointers that point
to the affected tree node are passed in and out.
*****************************************************************************/
double_right (head, stack, critical_node)
STRUCT tree_node_rec *stack[], **head;
INT    critical_node;
\{
   INT pivot, zeroed,i;
   STRUCT tree_node_rec *pivot_left;
   STRUCT tree_node_rec *loc_stack[MAXSTACK];

   pivot = critical_node + 1;
   pivot_left = stack[pivot]->lpt;
   FOR (i = 0; i < MAXSTACK; i++)  loc_stack[i] = stack[i];
   IF (pivot == NULL)  zeroed = FALSE;
   ELSE IF ((pivot_left != NULL) && (pivot_left->tag == -1))
        zeroed = TRUE;
   ELSE  zeroed = FALSE;
   /* end if */
   stack[critical_node]->rpt = pivot_left;
   stack[pivot]->lpt = pivot_left->rpt;
   pivot_left->rpt  = stack[pivot];
   loc_stack[pivot]   = pivot_left;
   loc_stack[pivot+1] = stack[pivot];

   single_right (head, loc_stack, critical_node);

   IF ((stack[critical_node]->rpt != NULL) && (stack[critical_node]->lpt == NULL))
     stack[critical_node]->tag = 1;
   ELSE IF  ((stack[critical_node]->rpt == NULL) && (stack[critical_node]->lpt != NULL))
     stack[critical_node]->tag = -1;
   ELSE stack[critical_node]->tag = 0;
   /* end if */

   IF ((stack[pivot]->lpt == NULL) && (stack[pivot]->rpt != NULL))
     stack[pivot]->tag = 1;
   ELSE IF  ((stack[pivot]->tag == -1) && (stack[pivot]->rpt != NULL) &&
           (stack[pivot]->rpt->tag != 0))
     stack[pivot]->tag = 1;
   ELSE IF  ((stack[pivot]->lpt != NULL) && (stack[pivot]->rpt == NULL))
     stack[pivot]->tag = - 1;
   ELSE IF ((stack[pivot]->tag == -1) && (zeroed == TRUE) &&
           (stack[pivot]->rpt != NULL) && (stack[pivot]->rpt->tag == 0))
     stack[pivot]->tag = 1;
   ELSE  stack[pivot]->tag = 0;
   /* end if */
\}  /* end of double_right */
/****************************************************************************
This balance tree is called by the modify tag and then it will call the
the approprite rotation function to perform the balancing act.
*****************************************************************************/
balance_tree (head, process, stack, critical_node,global)
INT process, critical_node,global;
```

```
   STRUCT tree_node_rec *stack[], **head;
┌───\{ INT loc_cri, loc_node, son, grandson;
|    FILE *fout;
|
|    IF (global == TRUE) fout = fg;
|    ELSE fout = fl;
|    son = critical_node + 1;
|    grandson = critical_node + 2;
|    IF ((stack[critical_node]->lpt == stack[son]) &&
|          ┌───(stack[son]->lpt == stack[grandson])) \{
|          |  fprintf (fout, "single left rotation.\n\n");
|          |  single_left (head, stack, critical_node);
|    \}
|    ELSE IF ((stack[critical_node]->rpt == stack[son]) &&
|                 ┌───(stack[son]->rpt == stack[grandson])) \{
|        fprintf (fout, "single right rotation.\n\n");
|        single_right (head, stack, critical_node);
|    \}
|    ELSE IF ((stack[critical_node]->lpt == stack[son]) &&
|                 ┌───(stack[son]->rpt == stack[grandson]))\{
|        fprintf (fout, "double left rotation.\n\n");
|        double_left (head, stack, critical_node);
|    \}
|  ┌───ELSE \{
|  |      fprintf (fout, "double right rotation.\n\n");
|  |      double_right (head, stack, critical_node);
|  └───\}
|    /* end if */
|    IF ((process == DEL) && (critical_node > 1))
|       modify_tag (*head, process, &loc_cri, stack, critical_node, &loc_node);
|    /* end if */
|    RETURN;
└───\}  /* end of balance_tree */

   /**************************************************************************
   This deletion will remove the approprite tree node from the directory. First
   it use the string that pass in and call find node to find approprite location
   of the node in the directory. If it is found, it is then remove the node and
   call balance tree to rebalance the tree. This routine is useally called by
   the delete node in the main program.
   **************************************************************************/
   deletion (s,head,global)
   CHAR *s;
   STRUCT tree_node_rec **head;
   INT global;
┌───\{ STRUCT tree_node_rec *find_node(), *loc, *stack[MAXSTACK];
|    INT    critical, found, critical_node, stack_top,
|           bef_del, del_loc, bef_suc, suc, glo;
|    FILE *fout;
|
|    IF (global == TRUE) fout = fg;
|    ELSE fout = fl;
|    glo = global;
```

```
|   IF (*head == NULL) fprintf (fout, "Empty tree !!\n");
|  ┌──ELSE\{
|  |    loc = find_node (*head, s, &found, stack, &stack_top, 0);
|  |    /* print (s,global); */
|  |     IF (found != TRUE) fprintf (fout, " does not exit. Deletion denied !\n\n");
|  |     ┌──ELSE \{
|  |     |      fprintf (fout, " has been deleted and the tree is: \n");
|  |     |      IF ((stack[stack_top] == *head) && ((*head)->lpt == (*head)->rpt))
|  |     |           ┌──\{ free (*head);
|  |     |           |    *head = NULL;
|  |     |           |    fprintf (fout, "Empty Tree !!\n");
|  |     |           └──\}
|  |     |      ┌──ELSE IF ((stack[stack_top] == *head) && ((*head)->rpt == NULL))\{
|  |     |      |      free (stack[stack_top]);
|  |     |      |      *head = stack[stack_top]->lpt;
|  |     |      └──\}
|  |     |      ┌──ELSE \{
|  |     |      |      bef_del = stack_top - 1;
|  |     |      |      del_loc = stack_top;
|  |     |      |      loc = stack[del_loc]->rpt;
|  |     |      |      ┌──WHILE (loc != NULL) \{
|  |     |      |      |      stack_top++;
|  |     |      |      |      stack[stack_top] = loc;
|  |     |      |      |      loc = loc->lpt;
|  |     |      |      └──\}
|  |     |      |      suc     = stack_top;
|  |     |      |      bef_suc = stack_top - 1;
|  |     |      |      modify_tag (*head, DEL, &critical, stack, stack_top, &critical_node);
|  |     |      |      IF ((stack[del_loc]->rpt == NULL) && (stack[del_loc]->lpt == NULL))
|  |     |      |          ┌──\{ IF (strcmp(stack[bef_del]->info, stack[del_loc]->info) > 0)
|  |     |      |          |      stack[bef_del]->lpt = NULL;
|  |     |      |          |    ELSE stack[bef_del]->rpt = NULL;
|  |     |      |          |    free (stack[del_loc]);
|  |     |      |          └──\}
|  |     |      |      ELSE IF (stack[del_loc]->rpt == NULL)
|  |     |      |          ┌──\{ IF (strcmp(stack[bef_del]->info, stack[del_loc]->info) > 0)
|  |     |      |          |      stack[bef_del]->lpt = stack[del_loc]->lpt;
|  |     |      |          |    ELSE stack[bef_del]->rpt = stack[del_loc]->lpt;
|  |     |      |          |    free (stack[del_loc]);
|  |     |      |          └──\}
|  |     |      |      ┌──ELSE \{
|  |     |      |      |      strcpy (stack[del_loc]->info, stack[suc]->info);
|  |     |      |      |      stack[del_loc]->fnum = stack[suc]->fnum;
|  |     |      |      |      stack[del_loc]->ownerpt = stack[suc]->ownerpt;
|  |     |      |      |      IF (strcmp(stack[bef_suc]->info, stack[suc]->info) > 0)
|  |     |      |      |        stack[bef_suc]->lpt = stack[suc]->rpt;
|  |     |      |      |      ELSE  stack[bef_suc]->rpt = stack[suc]->rpt;
|  |     |      |      |      free (stack[suc]);
|  |     |      |      └──────\}
|  |     |  IF (critical == TRUE) \{
|  |     |      | IF ((stack_top - critical_node) < 3)\{
|  |     |      | IF (strcmp(stack[critical_node]->info, stack[critical_node+1]->info) > 0)
|  |     |      |  \{
```

```
|   |           |           |   IF (stack[critical_node]->rpt != NULL)
|   |           |           |      \{ stack[critical_node+1] = stack[critical_node]->rpt;
|   |           |           |         IF (stack[critical_node+1]->tag == 1)
|   |           |           |            stack[critical_node+2] = stack[critical_node+1]->rpt;
|   |           |           |         ELSE IF (stack[critical_node+1]->tag == -1)
|   |           |           |             stack[critical_node+2] = stack[critical_node+1]->lpt;
|   |           |           |         ELSE \{
|   |           |           |                IF (stack[critical_node+1]->rpt != NULL)
|   |           |           |                   stack[critical_node+2] = stack[critical_node+1]->rpt;
|   |           |           |                   ELSE IF (stack[critical_node+1]->lpt != NULL)
|   |           |           |                   stack[critical_node+2] = stack[critical_node+1]->lpt;
|   |           |           |               \}
|   |           |           |        \}
|   |           |           |   \}
|   |           |           | ELSE
|   |           |           |      \{ IF (stack[critical_node]->lpt != NULL)
|   |           |           |         \{ stack[critical_node+1] = stack[critical_node]->lpt;
|   |           |           |            IF (stack[critical_node+1]->tag == 1)
|   |           |           |               stack[critical_node+2] = stack[critical_node+1]->rpt;
|   |           |           |            ELSE IF (stack[critical_node+1]->tag == -1)
|   |           |           |               stack[critical_node+2] = stack[critical_node+1]->lpt;
|   |           |           |            ELSE
|   |           |           |               \{ IF (stack[critical_node+1]->lpt != NULL)
|   |           |           |                 stack[critical_node+2] = stack[critical_node+1]->lpt;
|   |           |           |                 ELSE IF (stack[critical_node+1]->rpt != NULL)
|   |           |           |                 stack[critical_node+2] = stack[critical_node+1]->rpt;
|   |           |           |               \}
|   |           |           |            \}
|   |           |           |         \}
|   |           |           |      \}
|   |           |           |      balance_tree (head, DEL, stack, critical_node,glo);
|   |           |           |      \}
|   |           |           └──────────\}
|   |           |        print_tree (0, *head,glo);
|   |           |        fprintf (fout, "\n");
|   |           └──────────\}
|   └──────────\}
|    RETURN;
└───\}  /* end of deletion */
/*******************************************************/
print_tree (num_blank, tree_node,global)
INT num_blank;
STRUCT tree_node_rec *tree_node;
INT global;
┌───\{
|    INT i, loc;
|    FILE *fout;
|
|    IF (global == TRUE) fout = fg;
|    ELSE fout = fl;
|
|    loc = global;
|    IF (tree_node != NULL)
```

```
|        ┌───\{ print_tree (num_blank + 10, tree_node->rpt,loc);
|        |    FOR (i = 1; i <= num_blank; ++i)
|        |        fprintf (fout, "%c", BLANK);
|        |     print (tree_node->info,loc);
|        |     print_tree (num_blank + 10, tree_node->lpt,loc);
|        └───\}
└───\}   /* end of print_tree */
   /**************************************************/
   This mul.c is stored as another file and it will be linked to main.c.
   The purpose of this multiplication function is to simulate the binary
   working of the actual hardware multiplication during the calculation of the
   system key using the Chinese Remainder Theorem.
   **********************************************************************/
   #include "header.h"
   CHAR *mul(om,oq)
   CHAR *om,*oq;
┌───\{
|    CHAR a[MAXIMUM],q1,result[MAXIMUM],com_m[MAXIMUM],carry,*q,*m;
|    REGISTER INT cycle,indx,i,j,k,indxm,indxq;
|
|    q = oq; m = om;
|    indxm = indxq = 0;
|    indxm = strlen(m); indxq = strlen(q);
|    indxm -= 1; indxq -= 1;
|   ┌───IF (indxm != indxq) \{
|   |    ┌───IF (indxq > indxm) \{
|   |    |        IF ((indxq+1) > MAXIMUM ) \{ printf("number in div is too large \n"); exit(0); \}
|   |    |        k = indxq; m[k+1] = '\0';
|   |    |         ┌───FOR (i=indxm; i>=1; i--) \{
|   |    |         |       m[k] = m[i];  k--;
|   |    |         └───\}
|   |    |        FOR (i= (indxq-indxm); i>=1; i--) m[i] = '0';
|   |    |        indx = indxq;
|   |    └───\}
|   |    ┌───ELSE IF (indxm > indxq) \{
|   |    |         ┌───IF ((indxm+1) > MAXIMUM ) \{
|   |    |         └───printf("number in div is too large \n"); exit(0); \}
|   |    |        k = indxm; q[k+1] = '\0';
|   |    |         ┌───FOR (i=indxq; i>=1; i--) \{
|   |    |         |      q[k] = q[i]; k--;
|   |    |         └───\}
|   |    |        FOR (i=(indxm-indxq); i>=1; i--) q[i] = '0';
|   |    |        indx = indxm;
|   |    └───\}
|   └───\}
|    ELSE indx = indxm;
|    com_m[indx+1] = '\0';
|    FOR (i=0; i<=indx; i++) a[i] ='0';
|        a[i] = '\0';
|   ┌───FOR (j=indx; j>=0; j--) \{
|   |       IF (m[j] == '0') com_m[j] = '0';
|   |       ELSE BREAK;
|   └───\}
```

```
|    com_m[j] = '1';
|   ┌──FOR (k=j-1; k>=0; k--) \{
|   |      IF (m[k] == '1') com_m[k] = '0';
|   |      ELSE com_m[k] = '1';
|   └──\}
|    q1 = '0'; FOR (i=0; i<=indx; i++) a[i] = '0';
|    a[i] = '\0';
|    FOR (cycle=0; cycle<= indx; cycle++)
|   ┌──\{
|   |     IF (q[indx] == q1)   /* either 11 or 00 */
|   |  ┌──\{
|   |  |    q1 = q[indx];
|   |  |    FOR (j=indx; j>=1; j--)  q[j] = q[j-1];
|   |  |    q[0] = a[indx];
|   |  |    FOR (j=indx; j>=1; j--) a[j] = a[j-1];
|   |  |    IF (a[1] =='1') a[0] = '1'; ELSE a[0] = '0';
|   |  └──\}
|   |  ┌──ELSE \{
|   |  |          ┌──IF (q[indx] == '0')  \{  /*** case of 01 ***/
|   |  |          |  result[indx+1] = '\0'; carry = '0';
|   |  |          | ┌─FOR (i=indx; i>=0; i--) \{
|   |  |          | |     ┌──IF (a[i] != m[i]) \{
|   |  |          | |     |     IF (carry == '0') \{ carry = '0'; result[i]='1'; \}
|   |  |          | |     |     ELSE \{ carry = '1'; result[i] = '0'; \}
|   |  |          | |     └──\}
|   |  |          | |     ┌──ELSE \{
|   |  |          | |     |        IF (a[i] == '1' && carry == '1') \{ carry = '1'; result[i]= '1';

|   |  |          | |     |        ┌──ELSE IF (a[i] == '1' && carry == '0') \{
|   |  |          | |     |        |         carry = '1'; result[i] ='0';
|   |  |          | |     |        └──\}
|   |  |          | |     |        ┌──ELSE IF (a[i] == '0' && carry == '1') \{
|   |  |          | |     |        |         carry = '0'; result[i] = '1';
|   |  |          | |     |        └──\}
|   |  |          | |     |        ┌──ELSE IF (a[i] == '0' && carry == '0') \{
|   |  |          | |     |        |         carry = '0'; result[i] = '0';
|   |  |          | |     |        └──\}
|   |  |          | |     └──\}
|   |  |          | └─\}
|   |  |      \} /**** if loop of  01 ***/
|   |  |     ┌──ELSE \{
|   |  |     |           result[indx+1] = '\0'; carry = '0';
|   |  |     |     ┌──FOR (i=indx; i>=0; i--) \{
|   |  |     |     |      ┌──IF(a[i] != com_m[i]) \{
|   |  |     |     |      |     IF (carry == '0') \{ carry = '0'; result[i] = '1'; \}
|   |  |     |     |      |     ELSE \{  carry = '1'; result[i] = '0'; \}
|   |  |     |     |      └──\}
|   |  |     |     |      ┌──ELSE \{
|   |  |     |     |      |        ┌──IF (a[i] == '1' && carry == '1') \{
|   |  |     |     |      |        |     carry = '1'; result[i] ='1';
|   |  |     |     |      |        └──\}
|   |  |     |     |      |        ┌──ELSE IF  (a[i] == '1' && carry == '0') \{
|   |  |     |     |      |        |         carry = '1'; result[i] ='0';
|   |  |     |     |      |        └──\}
```

```
|   |   |       |         |         |             ┌──ELSE IF (a[i] == '0' && carry == '1')
|   |   |       |         |         |             |        carry = '0'; result[i] = '1';
|   |   |       |         |         |             └──\}
|   |   |       |         |         |             ┌──ELSE IF (a[i] == '0' && carry == '0') \{
|   |   |       |         |         |             |        carry = '0'; result[i] = '0';
|   |   |       |         |         |             └──\}
|   |   |       |         |         └───\}
|   |   |       |         └───\}
|   |   |       |              /*** case of 10 ***/
|   |   |       |            /*  printf("****result of addition with A - M is %s\n",result);
|   |   |       |             */
|   |   |       └───\}
|   |   |          strcpy(a,result);
|   |   |          q1 = q[indx];
|   |   |          FOR (j=indx; j>=1; j--) q[j] = q[j-1];
|   |   |          q[0] = a[indx];
|   |   |          FOR (j=indx; j>=1; j--) a[j] = a[j-1];
|   |   |          IF (a[1] == '1') a[0] = '1'; ELSE a[0] = '0';
|   |   └───\}  /** end of else for case 10 or 01 *****/
|   └───\} /* end of the cycle */
|          strcpy(result,a); strcat(result,q);
|            ┌──FOR (j=0; j<=(2*indx); j++) \{
|            |       IF (result[j] == '1') BREAK;
|            └───\}
|          result[0] = '0'; k =1;
|            ┌──FOR(i=j; i<=(2*indx)+1; i++) \{
|            |      result[k] = result[i];
|            |      k++;
|            └───\}
|          result[k] = '\0';
|          RETURN(result);
|
|   /*****************************************************************************/
|   This binary division is to simulate the binary division inside the hardware
|   of the system. It could be called by any function and the m is the multiplier
|   and the q is the quotien. The needq is the flag that tells this subroutine
|   that whether the remainder or the quotient is needed.
|   /*****************************************************************************/
|   #include "header.h"
|   CHAR *bdiv(m,q,needq)
|   CHAR *m,*q;
|   INT needq;
| ┌─\{
| |   CHAR a[MAXIMUM],result[MAXIMUM],com_m[MAXIMUM],sign,carry;
| |   REGISTER INT i,cycle,j,k;
| |   INT indxq,indxm,indx;
| |   indxq = strlen(q); indxm = strlen(m);
| |   indxq -= 1; indxm  -= 1;
| | ┌──IF (indxm != indxq) \{
| | |   ┌──IF (indxq > indxm) \{
| | |   |       IF ((indxq+1) > MAXIMUM ) \{ printf("number in div is too large \n"); exit(0); \}
| | |   |       k = indxq; m[k+1] = '\0';
| | |   |        ┌──FOR (i=indxm; i>=1; i--) \{
```

```
| | |    |          |        m[k] = m[i];  k--;
| | |    |          └───\}
| | |    |        FOR (i= (indxq-indxm); i>=1; i--) m[i] = '0';
| | |    |        indx = indxq;
| | |    └───\}
| | |    ┌──ELSE IF (indxm > indxq) \{
| | |    |        IF ((indxm+1) > MAXIMUM ) \{ printf("number in div is too large \n"); exit(0); \}
| | |    |        k = indxm; q[k+1] = '\0';
| | |    |         ┌──FOR (i=indxq; i>=1; i--) \{
| | |    |         |      q[k] = q[i]; k--;
| | |    |         └───\}
| | |    |        FOR (i=(indxm-indxq); i>=1; i--) q[i] = '0';
| | |    |        indx = indxm;
| | |    └───\}
| | └───\}
| |   ELSE indx = indxm;
| |   com_m[indx+1] = '\0';
| |   FOR (i=0; i<=indx; i++) a[i] ='0';
| |       a[i] = '\0';
| | ┌──FOR (j=indx; j>=0; j--) \{
| | |      IF (m[j] == '0') com_m[j] = '0';
| | |      ELSE BREAK;
| | └───\}
| |   com_m[j] = '1';
| | ┌──FOR (k=j-1; k>=0; k--) \{
| | |      IF (m[k] == '1') com_m[k] = '0';
| | |      ELSE com_m[k] = '1';
| | └───\}
| | ┌──FOR (cycle =0; cycle<=indx; cycle++)   \{
| | |     sign = a[0];
| | |     FOR (i=0; i<=indx-1; i++) a[i] = a[i+1]; /* shift left */
| | |     a[indx] = q[0];    /* shift left for A */
| | |     FOR (i=0; i <=indx-1; i++) q[i] = q[i+1];   /* shift left for Q */
| | |    ┌──IF (a[0] != m[0]) \{
| | |    |        result[indx+1] = '\0'; carry = '0';
| | |    |         ┌──FOR (i=indx; i>=0; i--) \{
| | |    |         |       ┌──IF (a[i] != m[i]) \{
| | |    |         |       |     IF (carry == '0') \{ carry = '0'; result[i]='1'; \}
| | |    |         |       |     ELSE \{ carry = '1'; result[i] = '0'; \}
| | |    |         |       └───\}
| | |    |         |       ┌──ELSE \{
| | |    |         |       |        IF (a[i] == '1' && carry == '1') \{ carry = '1'; result[i]= '1';
| | |    |         |       |           ┌──ELSE IF (a[i] == '1' && carry == '0') \{
| | |    |         |       |           |          carry = '1'; result[i] ='0';
| | |    |         |       |           └───\}
| | |    |         |       |           ┌──ELSE IF (a[i] == '0' && carry == '1') \{
| | |    |         |       |           |          carry = '0'; result[i] = '1';
| | |    |         |       |           └───\}
| | |    |         |       |           ┌──ELSE IF (a[i] == '0' && carry == '0') \{
| | |    |         |       |           |          carry = '0'; result[i] = '0';
| | |    |         |       |           └───\}
| | |    |         |       └───\}
| | |    |         └───\}
```

```
| | |     └───\}
| | |     ┌───ELSE \{
| | |     |          result[indx+1] = '\0'; carry = '0';
| | |     |          ┌───FOR (i=indx; i>=0; i--) \{
| | |     |          |       ┌───IF(a[i] != com_m[i]) \{
| | |     |          |       |      IF (carry == '0') \{ carry = '0'; result[i] = '1'; \}
| | |     |          |       |      ELSE \{  carry = '1'; result[i] = '0'; \}
| | |     |          |       └───\}
| | |     |          |       ┌───ELSE \{
| | |     |          |       |          ┌───IF (a[i] == '1' && carry == '1') \{
| | |     |          |       |          |     carry = '1'; result[i] ='1';
| | |     |          |       |          └───\}
| | |     |          |       |          ┌───ELSE IF  (a[i] == '1' && carry == '0') \{
| | |     |          |       |          |        carry = '1'; result[i] ='0';
| | |     |          |       |          └───\}
| | |     |          |       |          ┌───ELSE IF (a[i] == '0' && carry == '1') \{
| | |     |          |       |          |        carry = '0'; result[i] = '1';
| | |     |          |       |          └───\}
| | |     |          |       |          ┌───ELSE IF (a[i] == '0' && carry == '0') \{
| | |     |          |       |          |        carry = '0'; result[i] = '0';
| | |     |          |       |          └───\}
| | |     |          |       └───\}
| | |     |          └───\}
| | |     └───\}
| | |     ┌───IF (sign == result[0]) \{  /* successful */
| | |     |    q[indx] = '1';
| | |     |    strcpy(a,result);
| | |     └───\}
| | |     ┌───ELSE \{             /* unsuccessful */
| | |     |        q[indx] = '0';
| | |     └───\}
| | └───\}
| | ┌───IF (needq == 1) \{
| | |     strcpy(result,q);
| | |     FOR (j=0; j<=indx; j++) \{ IF (result[j] == '1') BREAK;  \}
| | |     result[0] = '0'; k=1;
| | |     FOR (i=j; i<=indx+1; i++) \{ result[k] = result[i]; k++; \}
| | |     result[k] = '\0';
| | |     RETURN(result);
| | └─────\}
| | ┌───ELSE \{ strcpy(result,a);
| | |        FOR (j=0; j<=indx; j++) \{ IF (result[j] == '1') BREAK; \}
| | |        result[0] = '0'; k=1;
| | |        FOR (i=j; i<=indx+1; i++) \{ result[k] = result[i]; k++; \}
| | |        result[k] = '\0';
| | |        RETURN(result);
| | └─────\}
| └──\}
|  /***************************************************************************/
|  This add routine will simulate the hardware addition of the binary calculation
|  The input are the pointers to string and it will RETURN the pointer to string
|  FOR the calling function.
|  ***************************************************************************/
```

```
|  #include "header.h"
|  CHAR *add(a,m)
|  CHAR *a,*m;
| ┌─\{
| |   CHAR result[MAXIMUM],carry;
| |   REGISTER INT indx,i,k,indxm,indxa;
| |
| |   indxm = strlen(m); indxa = strlen(a);
| |   indxm -= 1; indxa -= 1;
| |
| | ┌──IF (indxm != indxa) \{
| | |    ┌──IF (indxa > indxm) \{
| | |    |         ┌──IF ((indxa+1) > MAXIMUM ) \{
| | |    |         └──printf("number in div is too large \n"); exit(0); \}
| | |    |        k = indxa; m[k+1] = '\0';
| | |    |         ┌──FOR (i=indxm; i>=1; i--) \{
| | |    |         |       m[k] = m[i];  k--;
| | |    |         └──\}
| | |    |        FOR (i= (indxa-indxm); i>=1; i--) m[i] = '0';
| | |    |        indx = indxa;
| | |    └──\}
| | |    ┌──ELSE IF (indxm > indxa) \{
| | |    |         ┌──IF ((indxm+1) > MAXIMUM ) \{
| | |    |         └──printf("number in div is too large \n"); exit(0); \}
| | |    |        k = indxm; a[k+1] = '\0';
| | |    |         ┌──FOR (i=indxa; i>=1; i--) \{
| | |    |         |      a[k] = a[i]; k--;
| | |    |         └──\}
| | |    |        FOR (i=(indxm-indxa); i>=1; i--) a[i] = '0';
| | |    |      /*  printf("A --> %s and M --> %s \n",a,m);
| | |    |        */
| | |    |        indx = indxm;
| | |    └──\}
| | └──\}
| |   ELSE indx = indxm;
| |
| |   result[indx+1] = '\0'; carry = '0';
| | ┌──FOR (i=indx; i>=0; i--) \{
| | |          ┌──IF (a[i] != m[i]) \{
| | |          |     IF (carry == '0') \{ carry = '0'; result[i]='1'; \}
| | |          |     ELSE \{ carry = '1'; result[i] = '0'; \}
| | |          └──\}
| | |          ┌──ELSE \{
| | |          |       IF (a[i] == '1' && carry == '1') \{ carry = '1'; result[i]= '1'; \}
| | |          |          ┌──ELSE IF (a[i] == '1' && carry == '0') \{
| | |          |          |          carry = '1'; result[i] ='0';
| | |          |          └──\}
| | |          |          ┌──ELSE IF (a[i] == '0' && carry == '1') \{
| | |          |          |          carry = '0'; result[i] = '1';
| | |          |          └──\}
| | |          |          ┌──ELSE IF (a[i] == '0' && carry == '0') \{
| | |          |          |          carry = '0'; result[i] = '0';
| | |          |          └──\}
```

```
| | |          └───\}
| | └───\}
| |  /* printf("result of addition before moving %s \n",result);
| |        */
| |           ┌──IF (result[0] == '1') \{
| |           |     result[indx+2] = '\0';
| |           |     FOR (i = indx; i>=0; i--)    result[i+1] = result[i];
| |           |     result[0] = '0';
| |           └───\}
| | /*      printf("result of addition --> %s \n",result);
| |         */
| |         RETURN(result);
| └──\}
|   /*********************************************************************
|   This binary to integer routine will receive the binary character from the
|   calling routine and convert it to unsigned long integer.
|   *********************************************************************/
|   UNSIGNED LONG INT bin2int(s)
|   CHAR *s;
| ┌──\{
| |   REGISTER INT i,j,len,time;
| |   UNSIGNED LONG INT locval,y;
| |
| |   len = strlen(s);   len -= 1;
| |   j =0; locval =0;
| |
| | ┌───FOR(i=len; i>=1; i--) \{
| | |      ┌───IF (s[i] == '1') \{
| | |      |      IF (j == 0) locval += 1;
| | |      |      ┌───ELSE \{
| | |      |      |        y =1;
| | |      |      |        FOR (time = 1; time<=j; time++)  y *= 2;
| | |      |      |        locval = locval + y;
| | |      |      └───\}
| | |      └───\}
| | |      j++;
| | └───\}
| |   RETURN(locval);
| └──\}
|   /*********************************************************************
|   This integer to binary function get integer input and convert it into
|   binary number in character and return them as a pointer to string.
|   *********************************************************************/
|   #include "header.h"
|   CHAR *int2bin(n)
|   INT n;
| ┌──\{
| |   CHAR s[MAXIMUM],str1[MAXIMUM];
| |   INT index,i,k;
| |
| |   i = 0;
| |   IF (n == 0) \{ printf("value send in is 0\n"); exit(0); \}
| | ┌───IF (n == 1) \{
```

```
| | |          s[0] ='0'; s[1] = '1'; s[2] = '\0';  RETURN(s);
| | └────\}
| | ┌────DO \{
| | |      IF( (n % 2) == 1)  s[i] = '1';
| | |      ELSE  s[i] = '0';
| | |      ┌───n = n /2;  i++;  IF (i == MAXIMUM) \{
| | |      └───printf("too large array in int2bin\n"); exit(0); \}
| | └────\} WHILE ( n != 1  );
| |
| |   s[i] = '1'; s[i+1] = '\0';  index = i; k= 1; str1[0] = '0';
| |   FOR (i = index; i>=0; i--) \{ str1[k] = s[i]; k++; \}
| |   str1[k] = '\0';  RETURN(str1);
| └──\}
|    /**********************************************************************/
|
|
```

VITA

Kim S. Lee

Candidate for the Degree of

Master of Science

Thesis: A HIERARCHICAL SINGLE-KEY-LOCK ACCESS CONTROL USING THE CHINESE REMAINDER THEOREM

Major Field: Computer Science

Biographical:

Personal Data: Born in Tapah, Perak, West Malaysia, September 11, the son of Choon Gan Lee and Ngan Siew Thong.

Education: Graduated from Monk's Hill Secondary School, Singapore, in December 1981; received Bachelor of Science Degree in Business Administration (Majoring in Accounting) from Oklahoma State University at Stillwater in May, 1988; completed requirements for the Master of Science degree at Oklahoma State University in December, 1991.

Professional Experience: Programmer Trainee, System Department of Ong's Construction Company, Singapore, January, 1982, to December, 1983; Junior Programmer, System Department, Hyatt Regency Hotel, Malaysia, January, 1984, to July, 1985; Student Programmer, Department of Agriculture Economics, Oklahoma State University, August, 1990, to December, 1991.