PRACTICAL CONCURRENCY CONSIDERATIONS FOR A STUDENT ORIENTED DATABASE MANAGEMENT SYSTEM

by

JULIE ANNE SHAMPER

B.Sc., The University of Alberta, 1975

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

September 1980

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date Sept. 19/80

ABSTRACT

The correctness (or integrity) of a database may be destroyed when the database is accessed concurrently by a number of independent transactions. Part of the job of a database management system is to control concurrency so that correctness is guaranteed. This thesis examines concurrency controls in general, and in particular those provided by the Educational Data Base System (EDBS). It is concluded that EDBS facilities are not sufficient to guarantee correctness. Several alternatives are proposed whereby correctness can be guaranteed. An approach is selected based on ease of implementation rather than on the degree of concurrency provided. Guaranteeing correctness and at the same time providing a high degree of concurrency turns out to be a very difficult problem both in theory and in practice.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I sincerely thank my supervisors Paul Gilmore and Al Fowler for their careful review and perceptive criticism of my work. I also wish to thank Bob Goldstein for his direction during the formative stages of the thesis, his continued support and careful reading of the final document. Finally, I wish to thank the computer center staff both at UBC and at SFU for their help in getting the Educational Data Base System operational.

Chapter I: Introduction

This thesis has two major aspects: concurrency and a student oriented database management system.

The student oriented database management system is the Educational Data Base System (EDBS). EDBS consists of a number of APL workspaces. Users access the system from an APL terminal to perform various operations on databases. Utilities are provided to CREATE, DESTROY, and MAINTAIN databases. Other commands permit a user to retrieve data from a database, update data contained in a database, insert new data into a database, or delete data from a database.

EDBS is intended for use in teaching database concepts. There are at least two possible approaches to using EDBS in an educational environment. First, each student could create and manipulate his own databases. Second, the instructor could create a common database to be accessed by all students.

Concurrency is defined as the simultaneous use of a system by more than one user. When a database system is used concurrently, data in a database may become incorrect. The objective of concurrency research is to provide concurrency while ensuring database correctness. The usual approach is to provide a concurrency component for a database management system which controls concurrency such that correctness is guaranteed.

A. Motivation

Concurrency became an issue in relation to EDBS during installation of the system at UBC. The available version of EDBS was dependent on the CDC/APL file system. Before the system could be installed, it had to be modified to rely on the available MTS/APL file system.

The two file systems differ primarily in their data sharing facilities (i.e., access authorization, concurrency control). The particular demands that EDBS makes of the file system in terms of access authorization could be met in a straight forward manner via the available MTS APL functions. It appeared impossible, however, to simulate required concurrency control capabilities. It thus became necessary to consider very carefully the issue of concurrency before continuing with the implementation.

B. Proposed Work

A major portion of this thesis is concerned with defining the problem. Two major constraints imposed on the problem are the following: First, we are working in an environment of imperfect information. In particular, it is very difficult to obtain required information about the existing EDBS concurrency solution. Second, our basic approach requires that EDBS should not be modified, thus limiting the array of practical solutions. A summary of proposed work follows:

(1) review the literature in the area of concurrency to clarify basic objectives of concurrency

control.

(2) examine EDBS concurrency control facilities to determine the extent to which the existing system achieves these objectives.

(3) examine other systems to determine usual solutions.

(4) propose a solution which satisfies the basic objectives of concurrency control.

C. Dimensions

This thesis has several dimensions which may be described as follows:

1. File Systems

EDBS has been previously modified to convert from a dependency on the APL*PLUS file system to a dependency on the CDC/APL file system. Our undertaking then is to convert from a dependency on the CDC/APL file system to a dependency on the available MTS/APL file system. In this thesis, as part of our problem definition, we examine EDBS concurrency solutions as they developed through each file system. This approach is motivated by a desire to understand first, what correctness guarantee was initially intended for EDBS and second, the extent to which the existing concurrency solution is constrained by the CDC/APL file system. Hence, one major theme of this thesis is file systems, in particular their concurrency control capabilities.

2. Data Models

EDBS supports creation of relational, hierarchical and network databases. In this thesis we review essential characteristics of each data model and we examine at least one system other than EDBS which supports the data model. Our analysis is aimed at gaining insight into the question of whether the data model implies a need for a particular type of concurrency solution. Hence, a second major theme of this thesis is data models, in particular their concurrency control requirements.

3. Correctness

Concurrency is not the only factor in a database environment which jeopardizes correctness. The integrity of a database may be destroyed due to hardware failures, human errors, or software errors. Many systems provide integrity assertions for checking the validity of database updates and most systems provide a set of support routines to detect and recover from error situations. In this thesis we attempt to deal with the issue of concurrency within the broader framework of correctness in general by reviewing additional procedures for ensuring correctness provided by each system and EDBS.

4. Trade offs

In this thesis we are interested in trade offs, two of which are particularly relevant: the trade off between information and performance as pointed out by Kung and Papadimitriou (1979) in relation to concurrency control mechanisms and the trade off between concurrency and correctness as evidenced by comparing concurrency solutions provided by EDBS and other systems.

C. Reader's Guide

Chapter II of this thesis reviews the literature and describes concurrency control facilities provided by a number of systems. Chapter III describes EDBS and compares its concurrency control facilities with those of other systems. Chapter III concludes with a statement of the EDBS concurrency problem. Chapter IV describes alternatives and selects what we believe to be an optimum approach. Finally, Chapter V points directions for further work.

The reader is advised that this thesis examines many issues not necessarily related to the final solution. This thesis is intended to reflect our experience with the EDBS undertaking, which was basically a search to determine the problem and find an implementable solution.

Chapter II: Framework

This chapter reviews research to date in the area of database concurrency. The objective of concurrency research is to provide concurrency while ensuring database correctness. One approach is to provide a "serializability" component for a data base management system which guarantees that concurrent transactions will have a serial view of the database. The performance of such a component is measured in terms of the amount of concurrency provided. Virtually all solutions proposed thus far have been based on locking techniques. It appears that locking unduly restricts the amount of concurrency which can be provided. Current research tends toward finding alternative mechanisms to guarantee serializability while providing a greater degree of concurrency.

Section 1 of this chapter describes basic concepts. Section 2 examines in detail the requirements for serializability as presented by Papadimitriou (1979). Section 3 examines locking protocols. Finally, section 4 reviews concurrency solutions as implemented in a number of systems.

A. Basic Concepts

A database is defined as a set of entities where each entity has a name and a value. Users execute actions on the database. For the purpose of our analysis we make the following assumptions about actions:

(1) Actions are executed indivisibly and one before the other (i.e., actions are executed with zero concurrency on a serial machine). In this sense we say that actions are atomic.

(2) The only types of actions are: read, write, create and destroy. Actions are performed on entities. We do not consider an action to be some computation involving only temporary variables. Rather, we view these sorts of computations as occurring indivisibly with the action.

A database represents facts about some portion of the real world. Hence, there exists a certain set of assertions (hereafter called consistency constraints) which necessarily must be satisfied for the database to be correct relative to the real world. For example, suppose that our database consists of the entities E1, E2,..., En. Consistency constraints applying to the database might be the following:

E1 is equal to E2

En is equal to n

E2 is the index of E3

A database is continuously undergoing changes as user actions transform it from one state to another. A database is said to be consistent if its current state satisfies the set of consistency constraints associated with the database .

Consistency is necessary but not sufficient for correctness. For example, if exactly one consistency constraint applies to the database that "E1<E2", then it is possible for a

user to erroneously update E2 without violating any consistency constraints. The database will be incorrect, however, because it does not accurately reflect the current state of that portion of the real world which it is intended to represent.

A database must be at least temporarily inconsistent during update of entities which are related to each other by some consistency constraint. For example, "in moving money from one bank account to another there will be an instant during which one account has been debited and the other not yet credited. This violates a constraint that the number of dollars in the system is constant".[1]

To guarantee correctness at some level, actions by the same user are grouped into units called transactions. A transaction, when executed alone, transforms a consistent database state into a new consistent state. Hence, a transaction serves as a unit of consistency.

In this section, we will follow a notation similar to that presented by Eswaran et al. (1976) for representing an action on a given entity by a particular transaction. For example, the read of entity E1 by transaction T1 will be represented as (T1,read,E1). The transaction name (T1 in this case) will be eliminated if it is clear from context which transaction is performing the action.

To clearly fix the idea of a transaction consider the following groups of actions, T1 and T2, and suppose that only one constraint applies to the database: that "E1 is equal to

---

[1] This passage is taken from page 624 of Eswaran et al. (1976).

E2".

T1 = {(read,E1), (read,E2), (write,E1),(write,E2)}

T2 = {(read,E1), (write,E1)}

In this example, T1 would be a transaction but T2 would not be a transaction. T1 is consistency preserving since it updates both E1 and E2 (assuming that the entities are updated correctly). Even though the database may be temporarily inconsistent during execution of T1, it will be consistent upon completion of T1. In contrast, T2 updates E1 but not E2. Hence, upon completion of T2 the constraint that E1 is equal to E2 is violated. T2 is not consistency preserving and therefore not a transaction.

Having grouped actions into transactions we want to run transactions concurrently by interleaving actions from different transactions. The resulting sequence of actions is called a schedule. We assume here that we have a scheduler whose input is a sequence of arriving transactions and whose output is the order of execution of actions within the transactions. In this environment interleaving of actions must be restricted such that each transaction is isolated from temporary inconsistencies induced in the database by other transactions.

To see the correctness problems associated with running transactions concurrently, consider our previous database to which the constraint applies that "E1 is equal to E2". Now, suppose that two transactions,

T1 = {(T1,write,E1), (T1,write,E2)}

T2 = {(T2,read,E1), (T2,read,E2)}

are run concurrently according to the following schedule:

{(T1,write,E1), (T2,read,E1), (T2,read,E2), (T1,write,E2)}

Note that this schedule gives T2 an inconsistent view of the database (i.e., T2 reads E1 not equal to E2 because T1 has updated E1 but not E2 at the time of reading). Now, if it had been intended that transaction T2 should update the same (or different) entities based on the inconsistent values read, then the database would be inconsistent upon completion of the schedule.

Another correctness problem is the classic "lost update problem" which occurs when two concurrent transactions read the same value for an entity and then try to write back an incremented value. The result is that the second update overwrites the first and one increment is lost. A database is certainly incorrect if updates get lost, and further it may be inconsistent if say two entities are related by a consistency constraint and an update gets lost for one entity but not the other.

If transactions are run sequentially (i.e., one after the other in some order) then the above correctness problems do not arise. The schedule resulting from sequential execution of a set of transactions is called a serial schedule. For example, given our previous transactions, T1 and T2, there are exactly two possible serial schedules:

SERIAL SCHEDULES

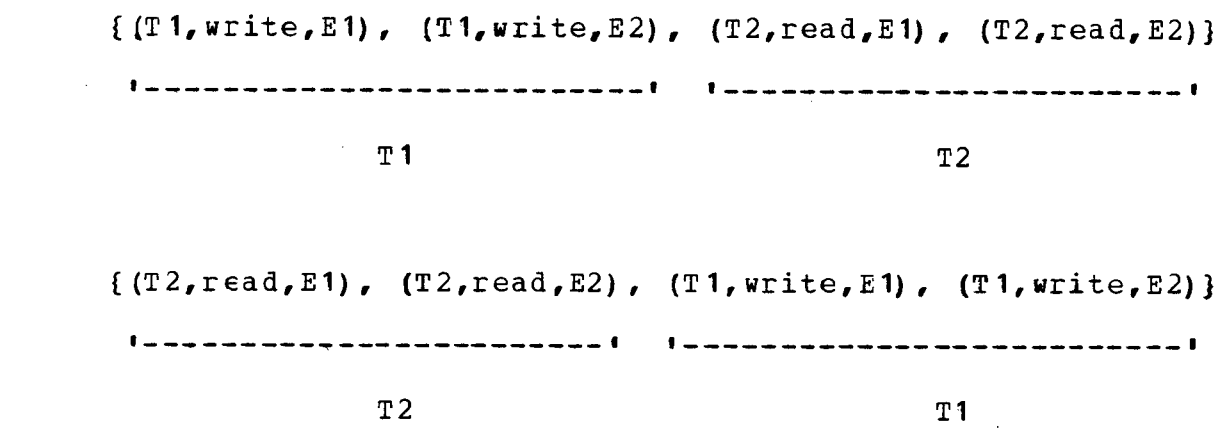

Figure 1

Serial schedules have a number of desirable properties. First, execution of transactions according to a serial schedule will give each transaction a consistent view of the database. This is because only one transaction will be active in the system at any one time. Second, the database will be consistent upon completion of a serial schedule because each transaction will transform a consistent database into a new consistent state. Third, the lost update problem cannot occur with a serial schedule because any read and write of an entity by a particular transaction will occur indivisibly. Of course, serial schedules do not provide concurrency between transactions (i.e., interleaving of actions from different transactions).

The usual approach (Eswaran et al. 1976, Papadimitriou 1979, Kedem and Silberschatz 1979) to providing concurrency while ensuring correctness involves the notion of serializability. A schedule is said to be serializable if it is equivalent to some serial schedule(equivalent in the sense that their outcomes are the same). The next section examines the notion of serializability as presented in Papadimitriou (1979).

B. Serializable Schedules

Papadimitriou justifies the appeal of serializability as a correctness criterion as follows:

> "Databases are supposed to be faithful models of parts of the real world, and user transactions represent instantaneous changes in the world. Since such changes are totally ordered by temporal priority, the only acceptable interleavings of atomic steps of different transactions are those that are equivalent to some sequential execution of these transactions."[1]

In this section, we describe Papadimitriou's model of transactions and characterization of schedule equivalence. The reader is cautioned that Papadimitriou considers a very narrow class of transactions. Namely, those that consist of exactly two actions: a read followed by a write. This simple transaction model is used in Papadimitriou (1979) as a framework for understanding and comparing different philosophies of serializability.[2] We present it here for similar reasons. We will see this model again in Chapter 3 of this thesis where it will serve as a means of describing essential characteristics of our EDBS concurrency problem.

Papadimitriou refers to a schedule as a history and views database entities as variables. Transactions are assumed to consist of two actions: the retrieval of the values of a set of variables followed by the update of the values of a set of variables. The read action for transaction Ti is denoted by Ri,

---

[1] This passage is taken from page 632 of Papadimitriou (1979).

[2] The same transaction model is used in Bernstein et al. (1978) and Rothnie and Goodman (1977).

the write action by Wi.

The set of variables read by a transaction is called the read set of the transaction (denoted by S(Ri) for transaction Ti). Similarily, the set of variables written, S(Wi), is called the write set of Ti. Each transaction is viewed as a set of uninterpreted function symbols. For example, suppose that S(Wi)={y} and S(Ri)={x,z}. Then we can view y as some function of x and z [i.e., y=fi(x,z)].

Consistent with our previous usage of the term, actions within transactions are assumed to be atomic. That is, we assume that values for variables in S(Ri) are read instantaneously. Similarily, we assume that values for variables in S(Wi) are written instantaneously.

Papadimitriou represents a history as follows: Suppose that the transactions involved in history h are T1 and T2 where S(R1)={x}, S(R2)={}, S(W1)=S(W2)={x,y}. Suppose also that actions from T1 and T2 are scheduled in the order {R1,R2,W1,W2}. Then h would be given by

h = R1[x]W1[x,y]W2[x,y][1]

Papadimitriou's notion of equivalence of histories is the following (Let V represent the set of variables in the database and let f(i,j) be the function associated with the jth variable in transaction Ti):

> "Two histories, h1 and h2, are equivalent if, given any set of |V| domains for the variables, any set of

---

[1] In contrast with Papadimitriou's approach we do not show in h read(write) actions for which the corresponding read(write) set is empty.

> initial values for the variables from the corresponding domains, and furthermore any interpretation of the functions f(i,j), the values of the variables are identical after execution of both histories."[1]

To understand Papadimitriou's syntactic characterization of history equivalence some terminology must first be introduced. In particular, given a history h we are concerned with the "augmented version of h", the "reads from" relation in h, and a "live" transaction in h.

The augmented version of h is the history h' derived from h by:

(1) appending onto the front of h, a transaction T which writes all variables in the database without reading any of them

(2) appending onto the end of h, a transaction T' which reads all variables in the database without updating any of them.

For example, if we let V be the set of variables in the database then h' would be represented as follows:

h' = W[V] ... { h } ... R'[V]

Now suppose that x S(Ri). A "reads from" relation in h is defined as follows:

> "Ri reads x from Wj in h if Wj is the latest occurrence of a write symbol before Ri in h' such that x ∈ S(Wj)"[2]

For example, given the augmented version of our previous

---

[1] This passage is taken from page 633 of Papadimitriou (1979).

[2] This passage is taken from page 634 of Papadimitriou (1979).

history,

$$h' = W[V]R1[x]W1[x,y]W2[x,y]R'[V]$$

we say that R1 reads x from W in h.

The definition of a live transaction in h is as follows:

(1) T' is live in h.

(2) If for some live transaction Tj, Rj reads a variable from Wi in h, then Ti is also live in h.

Papadimitriou showed that:

> "Two histories, h1 and h2, are equivalent if and only if they have the same set of live transactions and a live Ri reads x from Wj in h1 if and only if Ri reads x from Wj in h2."[1]

Recall that a serializable history is one which is equivalent to some serial history. Various notions of serializability exist in the literature. A major result presented in Papadimitriou (1979) is that recognizing transaction histories that are serializable is an NP-complete problem. In the same paper it is shown that previous notions of serializability actually provide for various subsets of the set of serializable histories.

The NP-complete result suggests that there is no efficient serializer of histories (i.e., algorithm which converts an input history to it's closest serializable history). However, Papadimitriou also shows that for all efficiently recognizable subclasses of serializable histories there exists an efficient scheduler for the class (i.e., an algorithm which transforms an input history into the closest serializable history within its

---

[1] This passage is taken from page 635 of Papadimitriou (1979).

class). In the next section, we will be concerned with these efficient schedulers.

C. Locking Protocols

There are certain difficulties in designing a scheduler in the form envisioned by Papadimitriou (1979). For example, the scheduler must be provided with syntactic descriptions of all transactions to be scheduled, and it is unclear to date how this might be accomplished. With the notable exception of the SDD-1 system (Bernstein et al. 1978) all concurrency solutions proposed thus far have been based on locking (i.e., semaphores controlling access to data). Transactions lock and unlock database entities according to a particular locking protocol. A locking protocol guarantees that if a transaction is run concurrently with any other set of transactions where each transaction follows the same locking protocol, any possible resulting schedule will be serializable. An important aspect of this approach is that it focuses on each transaction individually and hence the syntactic description of all transactions to occur in the history need not be known to the scheduler a priori. In this section, we discuss a number of issues related to locking protocols. We then describe two locking protocols in detail: two phase locking and the tree protocol. We conclude the section by relating a few practical considerations required in implementing a locking protocol.

The reader is advised that we are now considering a broader set of transactions than that considered by Papadimitriou. In this section, transactions may consist of more than two actions and except where otherwise indicated we include actions which create and destroy entities.

1. Modes of Locking

Various modes of locking have been proposed in the literature. A distinction between shared and exclusive access to an entity is often made. A shared lock is intended for reading an entity; an exclusive lock is intended if the entity is to be modified [see Kedem and Silberschatz (1979) for a definition of these two locking modes]. Gray et al. (1975) propose a third mode of locking called intention locking in which a transaction locks a given node in shared or exclusive mode by first locking all ancestors of the node in intention-shared or intention-exclusive mode. (Lockable resources in this context are assumed to form a directed acyclic graph.) Yannakakis et al. (1979) generalize locking to d-locking in which up to d-1 transactions may share an entity (i.e., a lock variable may assume d values 0, 1, ..., d-1 and the locked state is d-1).

In this section we will be concerned with exclusive locks the properties of which may be stated as follows:

(a) an entity may be locked by only one transaction at a time

(b) if a transaction Ti cannot lock an entity (because it is already locked by some other

transaction Tj) then Ti is suspended until Tj unlocks the entity.

2. Predicate Locks

Our previous discussion assumes that entities are accessed by name. However, in a database system it would be common for a transaction to want to access some logical subset of entities by specifying a condition on their values (i.e., key addressing). The requirement for accessing logical subsets of entities suggests the need for some form of logical locking, hereafter called predicate locking.

Predicate locks operate as follows (Schlageter 1978): Assume we have a set of entities described by predicate C which is to be locked. Then predicate C is noted down and each entity the value of which satisfies C is taken for locked.[1]

Predicate locking has been investigated by many researchers (Eswaran et al. 1976, Ries and Stonebraker 1977, Schlageter 1978). Further examination of this topic is beyond the scope of this thesis. We note only an example of what has been termed the "phantom problem" in the context of predicate locking.

---

[1] Schlageter (1978) also investigates the possibility of identifying the entities in the set defined by Predicate C and explicitly locking each such entity. The results are generally negative.

A phantom is an entity that really should not exist given the current database state. For example, if a transaction locks a set of entities as given by some predicate then no entities should be added to this set by another transaction until the first transaction terminates. Thus, the nonexistence of entities must also be locked. These nonexistent entities are called phantoms. The materialization of phantoms during the lifetime of a transaction may result in the transaction receiving an inconsistent view of the database. To illustrate this, consider the following example as described by Schlageter (1978).

> "A process P searches the same object set twice. The objects searched for in the first pass satisfy Predicate A, in the second pass Predicate A & B. As a result of phantom materialization the set found in the second pass may not be a subset of the set found in the first pass."[1]

3. Desirable Properties of a Locking Protocol

The performance of a locking protocol is evaluated in terms of the amount of concurrency that it provides. In Papadimitriou's terms, we could say that the larger the class of serializable schedules that are possible responses of a locking protocol, the greater the amount of concurrency provided (i.e., if we consider a locking protocol to be a scheduler then a larger class of recognizable serializable schedules would mean less reshuffling of the input schedule and fewer delays).

---

[1] This passage is taken from page 267 of Schlageter (1978). In our terminology substitute the terms transaction for process and entity for object.

Another desirable feature of a locking protocol, in addition to providing a large amount of concurrency, is that it ensures freedom from deadlock. Deadlock arises when say each of two transactions acquires only some of the entities that it needs and each waits for the other to unlock entities. The issue of deadlocks is examined in Yannakakis et al. (1979) for a subclass of locking policies called L-policies. It is shown that the problem of deciding whether a set of transactions is not deadlock free is NP complete, and that deadlock depends only on the order in which entities are locked (and not on where they are unlocked). In the next subsection we examine a locking protocol which is deadlock free and another which is not.

4. Two Phase Locking Protocol/Tree Protocol

In describing locking protocols we make the following assumptions:

(a) the locking or unlocking of an entity is an action

(b) a transaction locks each entity which is to be accessed prior to accessing it

(c) all locks are exclusive locks

(d) all actions (except lock and unlock) modify their entities (including read)

(e) entities are accessed by name

Further, we use the following model of serializability (Eswaran et al. 1976): Construct a directed graph D(S)

(a) with a node corresponding to each transaction Ti in Schedule S

(b) with an arc (Ti, Tj) whenever in S the output of some entity by Ti serves directly as input to Tj

To clarify (b) above, let A1 and A2 be actions and suppose that

S = {..., (Ti, A1, E), ..., (Tj, A2, E), ...}

Then the arc (Ti, Tj) would be found in D(S) if no other action involving entity E occurs between A1 and A2 in S. It can be proved (Eswaran et al. 1976) that if D(S) is acyclic then S is serializability.

Given a history of the form used in Papadimitriou (1979) let us construct the directed graph D(h). The nodes in D(h) would not necessarily correspond to live transactions in h. For example, transactions T1 and T2 are dead in h = R1[X]R2[X], but the arc (T1,T2) would appear in D(h).

Since Eswaran et al. assume that all actions are update actions, arc (T1,T2) will appear in the directed graph for each of the following histories:

h1 = R1[X]W2[X]

h2 = W1[X]R2[X]

h3 = W1[X]W2[X]

In both h1 and h3, output of X by T1 is said to serve as input to T2 even though in neither history is it the case that R2 reads X from W1. Note also that R2 reads X from W1 in h2 even though T2 is dead in h2.

The two phase lock (2PL) protocol proposed by Eswaran et al. (1976) states that once a transaction has unlocked some entity, it may not lock any more entities. Thus, a transaction has two phases: a locking phase and an unlocking phase. The following is an example of a transaction T1 which is 2PL and another T2 which is not 2PL:

T1 = {(lock, E1), (read, E1), (lock, E2)
(unlock, E1), (read, E2), (unlock, E2)}

T2 = {(lock, E1), (read, E1), (unlock, E1)
(lock, E2), (read, E2), (unlock, E2)}

T1 is 2PL because it locks all required entities before unlocking any. T2 is <u>not</u> 2PL because it requests a lock on entity E2 after releasing a lock on E1. Eswaran et al. (1976) have shown that 2PL is sufficient for ensuring serializability and also necessary in the sense that if a transaction, say Ti, does not follow 2PL, then there exists some Tj such that concurrent execution of the pair (Ti, Tj) may produce a non serializable schedule.

The two phase lock protocol does not ensure freedom from deadlock. To see this consider two transactions T1 and T2 where

T1 = {(lock, E1), (lock, E2),...,(unlock, E1), (unlock, E2)}

T2 = {(lock, E2), (lock, E1),...,(unlock, E2), (unlock, E1)}[1]

Note that both T1 and T2 are 2PL. Now, if T1 and T2 are run concurrently such that T1 succeeds in locking E1 and T2 succeeds in locking E2, then each transaction will wait indefinitely for the other to unlock its entity. Hence, T1 and T2 will be deadlocked.

How is it clear that 2PL ensures serializability? Ignoring deadlock, it is clear that D(S) will be acyclic for every possible schedule resulting from concurrent execution of a set of transactions T, if each transaction in T follows 2PL. This is because 2PL requires that a transaction must lock all its entities before releasing any. Thus, a situation could not occur with say two transactions in which Ti updates an entity before Tj does and Tj updates an entity before Ti does. Further, the arcs in D(S) represent a binary relation on the set of transactions in S and if this relation is acyclic then the transactions can be embedded in a linear order.

Although it has been shown that 2PL is essentially necessary for ensuring serializability when only syntactic information about the transaction system is available, other lock protocols have been derived which guarantee serializability by relying on information about database organization. Kedem and Silberschatz (1978, 1979) have proposed the Tree Protocol for databases organized as rooted trees and a DAG protocol for databases organized as directed acyclic graphs. The structure upon which these locking policies depend could in practice refer to some logical or physical organization of entities. For illustration

---

[1] As is usual we show only the lock and unlock steps.

purposes we focus here on the Tree protocol which may be stated as follows:

(a) A transaction may select the first entity in the database tree to be locked.

(b) Subsequently, the transaction may lock an entity only if it is currently holding a lock on the entity's father.

(c) A transaction may unlock an entity at any time, however a given entity may be locked at most once within the same transaction.

From (c) above it is clear that the tree protocol may result in transactions which are not two phase locked. Silberschatz and Kedem (1978) have shown that this protocol guarantees both serializability and freedom from deadlock.

5. Practical Considerations

When it is necessary for a transaction to lock entities it may in practice be more appropriate for the transaction to lock some larger granule of the database which contains the required entities. A granule in this context might be a record, a page, a file, a relation, a column of a relation, a tuple, etc. A small granule size allows a greater amount of concurrency at a greater cost in managing locks. A large granule size, on the other hand, inhibits concurrency but minimizes management of locks.

Ries and Stonebraker (1977) examine by simulation the overheads involved in locking. The results of this study show that a fairly large granule size is optimum in terms of such parameters as system throughout, response time, CPU utilization and I/O utilization. Their conclusion based on these results was that coarse granularity such as file or area locking is preferable to fine granularity such as page or record locking.

In a later study (Ries and Stonebraker 1979) several alternative assumptions in their model showed that the desirable granule size is more application dependent. For example, it was found that if all transactions are randomly accessing small parts of the database then fine granularity is preferable. However, if several transactions access large portions of the database then coarse granularity is again to be preferred.

These studies are of interest because they indicate that the cost of managing locks is high, and that any advantages due to greater concurrency may be outweighed by this cost.

Another motivation for locking, in addition to ensuring serializability is to facilitate transaction back out procedures. Back out may be necessary if it is found upon completion of a transaction that some consistency constraint has been violated or if deadlock is detected. Most systems maintain a log of all changes made to the database by each transaction. Back out of a transaction then, involves undoing all changes made by the transaction as recorded in the log. It is pointed out in Davies (1973) and Bjork (1973) that this procedure may not work correctly if a transaction is permitted to update uncommitted data. Uncommitted data is data which has been

updated by a transaction still in progress which may be later backed out.

To see the problem suppose that data updated by some ongoing transaction T1 is updated by another transaction T2. Then back out of transaction T1 will undo T2's update. Hence, T2 will also have to be backed out.

A need for "isolated" backout suggests that locks for update should be held to the end of a transaction. This is exactly the technique used in System R (Astrahan et al. 1976). Schlageter (1978) points out that the two phase lock protocol "prevents the propagation of rollback due to deadlock". Recall that two phase locking does not require that locks be held to the end of a transaction. Of course, propagation of rollback due to other reasons (i.e., hardware or software errors) is possible with two phase locking.

D. Other Systems

This section describes concurrency control facilities provided by a number of systems.

1. System R

System R is an experimental prototype database management system developed at the IBM San Jose, Research Laboratory (Astrahan et al. 1976). The system provides a high level relational data interface which permits definition of a variety of views on common underlying data. (The SEQUEL term for an external relation is a view.)

The principal language for interacting with System R is the SEQUEL data sublanguage described in Chamberlin and Boyce (1974). System R accomplishes the interface between SEQUEL and the host language by means of a concept called a cursor. A cursor is a name which identifies an active set of tuples and marks a particular tuple within this set. A cursor is positioned by means of the SEQUEL operator which takes as parameters a SEQUEL guery and a cursor name. The tuples within an active set may be retrieved one at a time by a user program by means of a FETCH command which takes as an argument a cursor name.

An alternative form of the FETCH command is provided. The FETCH-HOLD ccmmand operates in exactly the same way as the FETCH command except that it also acquires a hold on the tuple returned. A hold prevents other users from updating or deleting the tuple until it is explicitly released(by means of a RELEASE command) or until the holding transaction has terminated.

System R employs very sophisticated facilities for ensuring database correctness including integrity assertions, a logging and recovery subsystem, automatic locking, deadlock detection and transaction back out procedures.

Integrity assertions are specified by means of SEQUEL predicates. Assertions may describe permissible states of the database or permissible transitions in the database. The system automatically rejects any data modification command which violates an active integrity assertion. Assertions are normally checked at the end of each transaction, and the transaction is backed out if some assertion is violated. If an assertion is

specified as IMMEDIATE it is checked after each data modification command within each transaction.

In the remainder of this subsection we focus on the following concurrency control features of System R:

(a) user defined transactions

(b) multiple levels of consistency

(c) locking at various granularities

A user defines his own transactions in System R by means of the BEGIN-TRANS and END-TRANS operators. Such a transaction is basically a PL/1 program containing SEQUEL data manipulation commands. A user may also specify save points within his transaction. As long as a transaction is active the user may backup to the beginning or to any intermediate save point.

System R supports three distinct levels of consistency. When a user defines his transaction he must specify the level at which he wishes it to execute. At all three consistency levels the system guarantees that data modified by a transaction is not modified by any other until the first transaction terminates. Thus, the system can guarantee that backout of modifications by one transaction will not undo modifications made by other transactions. The differences in consistency levels occur during read operations.

Level 1 consistency offers the least isolation from other users. At this level a transaction may read uncommitted data. Such a transaction incurs the risk of reading inconsistent data values or data values which never existed if the transaction which set the values is later backed out. The possibly inaccurate data gathered by a level 1 transaction would be

unsuitable as a basis for updating the database or for making commitments to the outside world. It may be adequate, however, for statistical reporting. A level 1 transaction may explicitly employ the HOLD operator to protect itself from seeing uncommitted modifications or to guard against losing updates.

Level 2 consistency guarantees that a transaction does not see uncommitted data. However, the transaction may not see the same value for an entity each time the entity is accessed (i.e., read reproducibility is not guaranteed). At this consistency level it is possible for another transaction to modify an entity and commit the change in the interval between two successive accesses of the entity by a given level 2 transaction. Recall that data becomes committed as soon as the transaction which modifies it terminates. The HOLD operator may be used by a level 2 transaction to ensure read reproducibility and to guard against losing updates.

A level 3 transaction sees the logical equivalent of a single user system. All data read is committed and read reproducibility is guaranteed (except, of course, for changes made by the transaction itself). This read reproducibility applies not only to single tuples but also to collections of tuples. For example, if a level 3 transaction accesses all employees whose salary falls within a certain range, the same answer will occur every time within the transaction. Concurrent transactions will be prevented not only from updating or deleting an employee from this set but also from adding an employee to this set. Level 3 consistency does not require explicit HOLDs. The problem of lost updates is eliminated.

To guarantee consistency at these various levels, System R sets locks automatically (with the exception of explicit HOLDs which may be set by level 1 and 2 transactions). To reduce the overheads required for lock management, locks are applied at various granularities. For example, individual tuple locks are applied for transactions that access only a few tuples, whereas a coarser granularity of lock (i.e., at the level of a whole relation) may be chosen for transactions which access many tuples. The protocol for requesting locks is that described in Gray et al. (1976).

System R employs locking both at the logical level of relations and tuples and at the physical level of pages. At the physical level, locking is used to guard against occurrences such as the following:

> "... a data page may contain several tuples with each tuple accessed through its tuple identifier, which requires following a pointer within the data page. Even if no logical conflict occurs between two transactions because each is accessing a different relation or a different tuple in the same relation, a problem could occur at the physical level if one transaction follows a pointer to a tuple on some page while the other transaction updates a second tuple on the same page and causes a data compaction routine to reassign tuple locations."[1]

A distinction between shared and exclusive locks is made for logical locking. For all three consistency levels if a tuple is to be inserted or updated by a transaction then an exclusive lock is held on the tuple (or some superset) until the transaction terminates. Read reproducibility is achieved for level 3 transactions by maintaining shared locks on all tuples

---

[1] This passage is taken from page 124 of Astrahan et al. (1976).

and index values that are read for the duration of the transaction. "For transactions with level 2 consistency read accesses require a shared lock with immediate duration. Such a lock request is enqueued behind earlier exclusive lock requests so that the user is assured of reading committed data. The lock is then released as soon as the request has been granted, since reads do not have to be repeatable."[1] For transactions with level 1 consistency, no locks are held for read purposes except for physical locks on pages as described above.

2. INGRES

INGRES (Stonebraker et al. 1976) is another relational DBMS which, like System R, supports the concept of a view, provides integrity assertions for checking validity of updates, and provides a logging and recovery scheme. The concurrency control facilities provided by INGRES, however, are limited in comparison with those provided by System R, largely due to address space limitaticns of the PDP-11s for which INGRES was designed. INGRES provides an option whereby concurrency control can be turned off.

The primary query language supported is QUEL. Users may execute QUEL statements and other INGRES utilities directly from a terminal, or INGRES may be invoked from within a program written in EQUEL. EQUEL is a special language consisting of QUEL embedded in the general purpose programming language C.

---

[1] This passage is taken from page 126 of Astrahan et al. (1976).

Integrity assertions are specified by means of QUEL qualification clauses. A user's request is modified before execution by ANDing appropriate assertions with existing qualifications. This has the effect that no request can possibly violate any integrity assertion. INGRES does not support the System R transition assertions or deferred assertions.

The reader is cautioned that INGRES writers use the term transaction differently than we have been using it. INGRES transactions are not necessarily consistency preserving. Throughout the remainder of this subsection, whenever INGRES writers use the term transaction we use term INGRES transaction.

An INGRES transaction is defined as one INGRES command (i.e., QUEL data manipulation command or INGRES utility). The INGRES designers considered various other alternatives for defining a transaction including the following:

(a) a collection of INGRES commands with no intervening C code

(b) a collection of INGRES commands with C code but no system calls

(c) an arbitrary EQUEL program

Option (a) is perceived by INGRES designers to be a minor extension of the chosen alternative (one which they promised to implement given sufficient user demand). Option (b) is seen as a minor generalization of option (a) not worth the additional system complexity required for its implementation.

Defining an INGRES transaction as an EQUEL program [option (c)] is, in the opinion of the INGRES designers, impossible to

support. Such a transaction could contain system calls to say create and destroy files. The overhead of backing out through intermediate system calls to resolve deadlock is seen by INGRES designers as prohibitive. It is noted that deadlock cannot be avoided in this case because execution of a QUEL statement may depend on results obtained from system calls. Hence, there is no way to tell in advance that two transactions may deadlock. It is interesting to note that System R supports a transaction of this type, presumably because System R designers could better afford the overheads for resolution of deadlock (i.e., the operating system environment for System R is an extended version of VM/370).

INGRES designers have chosen to avoid (rather than detect and resolve) deadlock. The lock protocol used by INGRES transactions amounts to requiring that a transaction locks all its entities in one step. This is accomplished via a LOCK relation which records lock requests. An interaction[1] is not allowed to proceed unless all lock requests can be granted. (i.e., The lock relation is physically locked and interrogated for conflicting locks. If no locks conflict then all required locks are inserted into the LOCK relation. Otherwise, the interaction waits for a fixed interval and tries again.) It is not clear from available documentation whether both shared and exclusive locks are employed by the lock protocol. Locks are released at the end of each interaction.

---

[1] An interaction may consist of more than one INGRES transaction.

It is not clear what sort of correctness guarantee is offered by the INGRES lock protocol. If we assume that all locks are exclusive locks then it appears that the lock protocol will always output a serializable schedule when used by a set of consistency preserving transactions. When used by a set of INGRES transactions, the lock protocol may result in a schedule which is not consistency preserving. Clearly, any lock protocol even one that always outputs a serial schedule may result in an inconsistent schedule when used by non consistency preserving transactions.

It appears also that INGRES may lose updates. If there is no consistency preserving unit, then there is no way to require that the read and update of an entity by the same user be executed indivisibly. Hence, an update of the same entity by a different user may get interleaved such that an update gets lost.

We speculate that some protection against lost updates is offered by INGRES. First, since locks are held for the duration of an interaction, lost updates will be prevented if a user carefully selects the INGRES commands which will comprise his interaction. Second, it appears that INGRES automatically converts any update command into a retrieval and update command. This would mean that some lost update situations will be avoided by not allowing the update of a variable to be based on a previous and erroneous value read. It appears that the facility to process several commands as an interaction could be used in general to define transactions. These statements are highly speculative, however, as they have not been pointed out by

INGRES writers.

The granularity of lock is relatively coarse (on domains of a relation) reflecting an environment where core storage for a large lock table is not available. A predicate locking scheme is proposed in the hope that considerable concurrency may be provided at an acceptable overhead in lock table space and CPU time.

3. IMS

IMS is a hierarchical DBMS which provides facilities for running primarily batch applications. A user's external view consists of a collection of logical databases, each defined by means of a program control block (PCB) which also specifies a mapping to a corresponding physical database. When a user program is operating on a particular logical database the associated PCB is maintained in storage as a means of communication between the program and IMS.

Each physical database is an ordered set of all occurrences of one type of physical database record. The following describes essential features of the IMS hierarchical data model. The reader is referred to Date(1977) for further details:

(a) A database is defined by a tree structure (i.e., a definition tree). Each node of the definition tree represents a segment type in the database.

(b) The definition tree contains exactly one root segment type. The root may have any number of child segment types. Each child of the root may also have any number of child segment types and so

on to any number of levels.

(c) A database record consists of the occurrence in the database of a single root segment and a number of dependent segments.

(d) A database record follows the template imposed by the definition tree with the exception that for one occurrence of any given segment type there may be zero or more occurrences of each of its children.

IMS provides a complete set of routines to assist in maintaining database correctness including checkpoint and restart procedures, procedures for backing out changes made to a database by a given program, and procedures for maintaining the system log.

IMS does not explicitly provide integrity assertions. However, as pointed out in Date (1977), there are two features of IMS that can be viewed as mechanisms for handling consistency constraints. First, IMS enforces uniqueness of sequence field values (for segments which have sequence fields and are not multiple valued). Second, certain consistency constraints are enforced by the very hierarchical structure of an IMS database. For example, suppose that E1 and E2 are segment occurrences and that a consistency constraint C applies to the database that "E1 cannot exist without E2". If the database is organized such that the segment type of E2 is superior in the definition tree to that of E1, then C would be guaranteed by the simple rule for hierarchical databases that an occurrence of a dependent segment cannot exist without its parent.

IMS provides a fairly complete set of concurrency control facilities. The primary difference between IMS and other systems we have seen in this regard is that IMS leaves most of the responsibility for lock protocols up to the user (i.e., no automatic locking). The remainder of this section describes IMS concurrency control facilities.

First, a PCB associated with a user program may specify an option whereby all occurrences of an entire segment type are locked in exclusive mode. IMS will not load and initiate a program if its PCB entry conflicts with that of any program which is already executing. Two PCB entries conflict if either specifies the EXCLUSIVE option for the same segment type.

Second, IMS provides a hold mechanism to guard against lost updates. Two versions of each retrieval command are provided: a GET version and a GET-HOLD version. Both forms retrieve a particular segment occurrence, however a GET-HOLD command also places the retrieved segment in hold. When a segment is held by one user it may be retrieved by other users but not via a GET-HOLD retrieval command. When the holding user issues a modify command the segment cccurrence is locked in exclusive mode (i.e., other users may neither GET nor GET-HOLD the segment). Note that this procedure ensures that two users will not read the same value for a segment with the intention of updating it. Hence, segment updates cannot be overwritten.

Third, IMS supports shared locks on segment occurrences which may be explicitly set and released by a user program. Although there is a limit to the number of such locks that a program may hold at one time, this facility appears sufficient

to guarantee consistency (assuming judicious use of a locking protocol by all concurrent programs). It is interesting to note that IMS permits a user to delegate different segment occurrences to various lock classes. All segments within a given lock class may then be unlocked in one operation. The following is an example of such locking and unlocking:

```
GU PRESIDENT*QB(NAME='JOHN')
DEQ B
```

The '*Q' in the GET UNIQUE command specifies that the retrieved segment should be locked in shared mode. The 'B' following the *Q is the lock class. The dequeue command (DEQ) releases all shared locks on segments in lock class B. Note that a hold on a segment and shared lock on a segment are functionally identical.

Exclusive locks are released when a program terminates or when a checkpoint operation within the program is executed. Shared locks are also released at these times. Backout may be initiated by a program or by the system. The automatic unlocking strategy appears sufficient to guarantee isolated backout.

Finally, IMS permits specification of an option in the PCB (called the READ EXPRESS option) whereby a program is allowed to see uncommitted changes made by other concurrent programs. A program which uses the READ EXPRESS option would be known as a level 1 transaction in System R.

4. DBTG

In this section we review proposals made by the Data Base Task Group (DBTG) for maintaining correctness in a network based system (CODASYL, 1976). In particular, we are interested in the provisions for specifying integrity constraints in the schema DDL, and in the DML statements for concurrency control. We first briefly review the network data model and the fundamental notion of currency(not to be confused with concurrency).

A network is a more general structure than a hierarchy in that a given segment occurrence may have any number of immediate superiors (as well as any number of immediate dependents). The network approach uses the term record for segment and includes the concept of a set. A set type consists of one owner record type and one or more member record types. An occurrence of a set in the database consists of the occurrence of one owner record and zero or more member records. The reader is referred to Date (1977) for further details on the network data model.

The DBTG data sublanguage rests on the fundamental notion of currency. The basic idea is that the DBMS maintains for each active user program a table of database key values which specifies the most recently referenced record occurrence within each area,[1] within each record type, within each set type and within all record types. The most recently referenced record occurrence within all record types is called the current of run-unit.

---

[1] The storage space for a DBTG database is partitioned into a number of named areas.

The DBTG proposes specification of integrity constraints in the form of procedure declarations in the schema DDL. Such a procedure is automatically invoked before, after or on error during a particular operation affecting a particular object. The object may be the schema itself, an area, record, data item or set. Exactly when the procedure should be invoked as well as the object and operation that should trigger the invokation is also specified as part of the schema. Additional constraints may be specified which are automatically checked after execution of a store or modify operation involving a specified data item. This feature could be used, for example, to specify values or ranges of values allowed for data items.

The DBTG facilities for concurrency control are similar to those of IMS in that they are not applied automatically. Rather, it is left up to the user to develop his own concurrency control strategy based on the available concurrency control primitives. The remainder of this section describes two such concurrency control primitives: usage-mode and monitored-mode.

A user must specify a usage-mode for any area which he wishes to use. A user indicates two things when he specifies a usage-mode: First, how he himself wishes to use the area. Second, how he wishes concurrent users to use the area. The available usage-modes are described below:

(a) Retrieval - given user wishes to read from this area
- other users may read or write in this area

(b) Update - given user wishes to write in this area
- other users may read or write

(c) Protected Retrieval
- given user wishes to read from this area
- other users may read but not write

(d) Protected Update
- given user wishes to write in this area
- other users may read but not write

(e) Exclusive Retrieval
- given user wishes to read
- other users may neither read nor write

(f) Exclusive Update
- given user wishes to write in this area
- other users may neither read nor write

A user is not allowed access to an area if a conflicting usage-mode has already been established for that area by some other user. The facility for ensuring protected or exclusive use of an area might be effectively used to ensure database correctness. However, this mechanism severely restricts concurrency because the unit of locking (i.e., an area) is a large portion of the database.

The DBTG proposals provide for an additional mechanism which differs from any that we have seen so far in that, rather than isolating a user from the activities of other concurrent users, this mechanism attempts to notify a user of the activities of other concurrent users. The basic idea of the scheme is as follows: A user may explicitly place a record into monitored-mode. If a user, A, has a record in monitored-mode then an attempt by A to change the record will fail if some concurrent user has changed the record since A requested monitored-mode. User A receives an error message and it is up to him to decide what to do next. Monitored-mode has received criticism in the literature (Engles 1971, 1977). The basic problems are as follows:

(a) When a request to modify a record in monitored-mode fails, there is no way of forcing the user to

handle the situation correctly.

(b) No protection is offered against seeing uncommitted changes.

(c) No mechanism is provided to prevent concurrent users from changing data that a given user is currently working on.

(d) One user may transfer another user's current of run-unit from one record occurrence to another. (A possible result is that a user, unaware that his current of run-unit has been transferred, may find himself traversing the wrong set occurrence.)

Several proposals have been made to the CODASYL Programming Language Committee to replace the DBTG notify protocol with a locking scheme. One such proposal made by UNIVAC(1976) provides for a LOCK statement which permits a user to explicitly lock a record and keep it locked even when the record ceases to be the current of run-unit. [Some similar ideas were proposed earlier by Hawley et al. (1975).] A unified solution to several DBTG problems including concurrency has been proposed by Engles (1976) based on locking and the concept of a cursor.

E. Current Hot Research Issues

We conclude this chapter with a list of current hot research issues:

1. Quantitative measures for evaluating the performance of a scheduler are needed. To date only qualitative measures have been proposed such as the set of all output schedules. Papadimitriou (1979) suggests one promising direction which

would be to approximate the total number of delays imposed on requests by counting the number of transaction steps which cannot execute immediately upon arrival.

2. A basic assumption in Papadimitriou's formalism is that a syntactic description of all transactions to occur in a history is known to the scheduler a priori. In Papadimitriou's words "it is not clear how to remove this assumption and still retain the wealth of available solutions". One approach would be to have a number of prototype transactions to which arriving transactions can be matched. This approach is discussed in Papadimitriou (1979) and Bernstein et al. (1977). Recall that locking protocols get around this requirement by focusing on each individual transaction.

3. Many of the results presented in the literature related to locking strategies are not adequately generalized to distinguish between read and write actions. In general lock protocols are stated by assuming that all actions modify their entities. Some generalization of the two phase lock protocol can be found in Gray et al. (1975) and Lien and Weinberger (1978). Kedem and Silberschatz (1979), in the context of their work on lock protocols for database systems modeled by directed graphs, note a requirement for continued investigation "concerning correctness when transactions are permitted to set shared and exclusive locks".

4. A major current issue, not discussed anywhere else in this thesis except here, is distributed data management. For discussion purposes we choose the SDD-1 system - a prototype

distributed database system under development by Computer Corporation of America. The SDD-1 system involves redundant data stored at a number of network sites. Users interact with SDD-1 as if it were a non-distributed database system because SDD-1 handles all issues arising from distribution which, among many others, include distributed concurrency control. The concurrency issue here is the same as we have been describing except that it is further complicated by the problems of data distribution and data replication. A series of papers in ACM Transactions on Database Systems (1980) describes the SDD-1 system and its concurrency control mechanism. Further, an analysis of correctness of the concurrency control mechanism is provided (see Rothnie et al. 1980, Bernstein et al. 1980, Berstein and Shipman 1980). Many researchers have discussed extensions of their results to distributed databases (i.e., Papadimitriou 1979, Kedem and Silberschatz 1979).

5. Finally, there is the issue of concurrency in B-trees. A B-tree (Bayer and McCreight 1972) is a data structure for organizing a database to guarantee efficient access. Concurrent operation of processes on a database stored as a B-tree (or some variant) involves the usual problems associated with any concurrent system. However, specific solutions are suggested by the structure imposed by the B-tree and by a requirement for a limited set of operations. See Kwong and Wood (1979, 1980) for a survey of solutions to the problem of concurrency in B-trees and a discussion of the many remaining problems.

Chapter III: Problem

In general, our EDBS concurrency problem may be stated as follows: to ensure correctness of data maintained by EDBS when the system is used concurrently under MTS. This chapter provides a host of background material required to understand the problem. First, we describe EDBS from the user's point of view and we compare EDBS both in general and in terms of concurrency control to those systems which have been described in Chapter II. Second, we discuss EDBS implementation of concurrency control. This has several aspects. EDBS relies on the file system to do some of the locking required by its concurrency control mechanism. EDBS has now been implemented with three different file systems(i.e., APL*PLUS, CDC/APL, MTS/APL). Both the APL*PLUS and CDC/APL implementations are discussed to provide insight into the concurrency solution which was initially intended for EDBS. The existing solution may be constrained by the CDC/APL file system. It is our hope to devise a solution which is independent of previous file systems. Third, we describe EDBS concurrency control facilities using Papadimitriou's transaction model as presented in Chapter II. Finally, we provide a list of key problems related to running EDBS concurrently under MTS.

The reader is cautioned that in this chapter when we use the term EDBS we are referring to a set of APL functions which may be used to perform operations on databases.

A. EDBS Description

EDBS recognizes three types of users: system administrators, database administrators, and common users. A system administrator is responsible for installing, maintaining, and modifying EDBS. A database administrator (DBA) is responsible for creating, maintaining and granting access to one or more databases. A common user is one who has been given permission by a DBA to access a database. A DBA may also be a common user.

EDBS is comprised of a number of APL workspaces. One workspace contains all the utilities required by the system administrator to carry out his role (such as those required to install the system). Other workspaces are intended for use by DBAs and contain utilities for creating, destroying, granting access to, and maintaining databases.

All three data models -- hierarchical, network, and relational -- are supported by EDBS. A separate system (workspace) implements each data model. These workspaces are intended for common users. The reader is referred to Appendix A for an overview of EDBS workspaces and utilities.

Although there are many similarities among the commands available for interacting with the different types of databases, each system has its own data manipulation language. There are four categories of commands available with each system:

(1) Retrieval commands

(2) Modification commands

(3) Buffer commands

(4) Control commands

Data in a database is not manipulated directly by a user. Rather, it is moved into a buffer via retrieval commands where it may be read or modified via buffer commands. Modification commands take new or updated data from the buffer and place it in the database.

Common to all three systems is the concept of a HOLD, however the unit of holding is different for each system. As in other systems (System R, IMS) EDBS supports two versions of each retrieval command: GET and GET-HOLD. Both versions retrieve data. The GET-HOLD also acquires a hold on the data retrieved. If a hold is not immediately possible EDBS returns a status code to the user to advise him to try again later to hold the item.

There are three control commands: OPEN and CLOSE for opening and closing a database and RELEASE. The RELEASE command provides a user with the capability of explicitly releasing a hold which he has acquired by means of a previous GET-HOLD retrieval command.

EDBS does not provide integrity assertions. Two mechanisms are provided to recover from a system crash which may leave the database in an inconsistent state: The first is the MAINTAIN utility which massages the database into a consistent state through an interactive dialogue with the DBA. The second mechanism is the logging of all insertions, deletions and updates. A DUMPLOG utility is provided for printing the logged information. Logged entries may be erased via the ERASELOG

Utility.[1] The reader is referred to Appendix B for a summary of available EDBS data manipulation commands and utilities.

The objectives of this section are two-fold:

(1) to provide a flavour of how each system within EDBS differs from those we have seen before (both in general and in terms of concurrency control).

(2) to describe in detail the EDBS concurrency control facilities available at the user interface for each system.

As a means of comparison we choose System R for the EDBS relational system, IMS for the EDBS hierarchical system and DBTG for the EDBS network system.

1. Relational System

The EDBS relational system provides simple retrieval, qualified retrieval, and retrieval from more than one relation. In addition, a full range of storage operations are provided for inserting, updating, and deleting tuples. Among other things, the following capabilities of the SEQUEL data sublanguage are not supported by the EDBS relational DML:

(a) ordering on tuples retrieved

(b) nested mapping

(c) specification of more than two relations in a retrieval command

(d) explicit elimination of duplicates from a query

---

[1] The MAINTAIN, DUMPLOG and ERASELOG utilities have not been implemented at UBC.

result.

Like SEQUEL, all EDBS storage operations are restricted to operate on a single relation at a time.

The EDBS approach to defining new relations differs from that of System R. With System R facilities for data definition and data manipulation are provided in a unified manner by means of the SEQUEL operator. A user may create new base relations and destroy them when they are no longer needed. EDBS, on the other hand, distinguishes between data definition and data manipulation. A separate utility is provided for defining new base relations. Only a DBA (not a common user) may create and destroy base relations.

The method of interfacing the data sublanguage and the host language is similar with EDBS and System R. Recall that System R accomplishes the interface by means of cursors which identify active sets of tuples. The tuples may be read by a program by specifying the appropriate cursor in a FETCH command. EDBS maintains active sets of tuples in a buffer. Tuples are available to an APL program by means of buffer commands which require specification of a relation name. The relation name serves the function of a cursor.

The only concurrency control mechanism available with the EDBS relational system is the HOLD option. The System R automatic locking, multiple levels of consistency and transaction definition facilities are not supported. The HOLD option operates differently between EDBS and System R: The unit of hold with EDBS is one relation. The unit of hold with System R is a single tuple.

EDBS enforces certain rules on use of the GET-HOLD which may be stated as follows:

(a) A user must hold a relation prior to updating it.

(b) A user may hold at most one relation at one time.

(c) Only buffer commands (or the RELEASE command) may intervene between the GET-HOLD and UPDATE commands.

Prior use of the GET-HOLD is not required for insertion or deletion of tuples. EDBS automatically holds the relation in these instances.

2. Hierarchical System

The EDBS data description language for hierarchical databases is similar to that of IMS. The most notable difference is that EDBS does not support sequencing of key values or key values consisting of more than one field value. The EDBS data manipulation language for hierarchical databases is also similar to the IMS data sublanguage. The GET UNIQUE, GET NEXT, GET NEXT WITHIN PARENT, INSERT, DELETE, and REPLACE operations are all supported. These operations may be qualified by means of comparison and Boolean operators which comprise the equivalent of an IMS segment search argument.

In terms of concurrency control, only the HOLD option is supported by EDBS. The HOLD option operates exactly as described for IMS (see Chapter II). EDBS has no facility by which a user may hold more than one segment occurrence at one time. Recall that IMS provides this facility by means of the PCB exclusive option or by means of explicit shared locks.

EDBS enforces the following rules on use of GET-HOLD retrieval operations:

(a) Except for a root segment, the parent-to-be of any segment which is to be inserted must first be held via a GET-HOLD retrieval command.

(b) Any segment which is to be modified or deleted must first be held via a GET-HOLD retrieval command.

(c) Only buffer commands (or the RELEASE command) may intervene between a GET-HOLD retrieval command and any modification ccmmand.

3. Network System

The EDBS data description language for network databases is a rather limited version of the DBTG schema DDL. Many of the DBTG features concerned with the storage level of a database system are not supported by EDBS. Other DBTG options which may be specified in the schema are not supported by EDBS. For example, the storage class for every member record type in EDBS is assumed to be MANUAL. That is, when an occurrence of a member record type is first created and placed in the database, it is not automatically inserted as a member of any set occurrence. Furthermore, the removal class for every member record type in EDBS is always OPTIONAL. This means that an occurrence of any member record type may exist in the database without being a member of any set occurrence.

EDBS does not maintain as many currency indicators as DBTG. Since an EDBS database is not divided into named areas, there is no need to maintain the current record occurrence within each area. The most important DBTG currency of all -- the current of run-unit -- is not maintained by EDBS. This has obvious implications in terms of the respective DMLs. With DBTG virtually every operation uses the current of run-unit which is established via the FIND operation. With EDBS operations are performed on segment occurrences established by means of other currencies (i.e., record or set pointers).

The concurrency control facilities provided by the EDBS network system bear no resemblance to those proposed by the DBTG. Recall that the DBTG final report (1971) does not provide for use of holds. The remainder of this subsection describes the use of holds in the EDBS network system.

Five possible modification actions are available:

(a) insertion of a new record

(b) deletion of a record

(c) inclusion of a record in a set occurrence

(d) removal of a record in a set occurrence

(e) update of a record currently in the database.

The following rules apply to use of holds in the network system:

(a) Before a record may be updated it must be placed in hold. Upon completion of the update the hold is automatically removed.

(b) For INCLUDE and REMOVE the record may optionally be placed in hold. A RELEASE command must be

issued to release the hold.

(c) For deletion the record may optionally be placed in hold. The hold is automatically released upon completion of the DELETE command.

(d) A user may have at most one record occurrence in hold at one time.

There are certain difficulties inherent in the EDBS concurrency control scheme for network databases. As with DBTG the user is not completely isolated from the activities of other concurrent users. The EDBS User's Guide advises that "the existence of concurrent users necessitates that caution must be exercised when dealing with record or set pointers". A user may be affected in one of the following ways:

(a) A record or set pointer may mark the position of a record that has been deleted by another user. If the location remains empty and the pointer is referenced an error condition will result. If a new record is stored in the vacant location the prior deletion will be undetectable.

(b) A set pointer may mark the position of a member record that has been removed from a set occurrence by another user. If the pointer is referenced an error condition will result.

B. EDBS Implementation

EDBS was designed by the Systems Research Group at University of Toronto. The implementation was dependent on availability of the APL*PLUS file system and used overlay techniques to fit the system into a small workspace. The University of Calgary installed EDBS on their Control Data Computer (CDC). Overlay functions were eliminated and the system was converted to rely on the CDC/APL file system.

With the recent introduction of the MTS/APL file system it became feasible to install EDBS at UBC. The decision was to use the CDC/APL version of EDBS. The major requirement of the implementation was conversion of the file system. The major constraint was that the system itself should not be modified.

The general approach (suggested by Yair Wand, Faculty of Management, University of Calgary) was to provide an interface file system between EDBS and the MTS/APL file system. The interface file system is intended to simulate the CDC/APL file system using the available MTS/APL functions. The following diagram represents the relationship between EDBS, the interface file system and the MTS/APL file system.

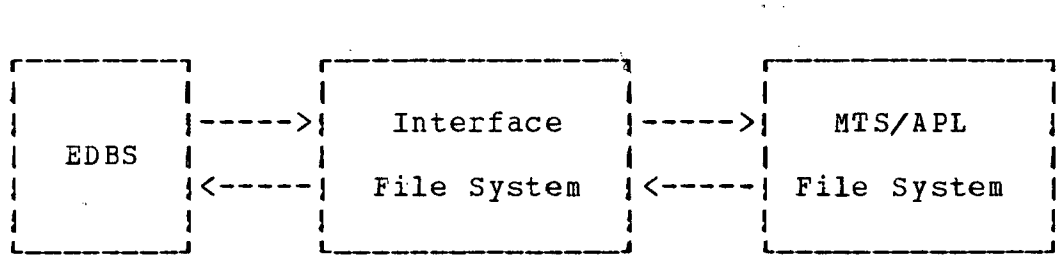

Figure 2

The functions available from the interface file system are exactly those provided by the CDC/APL file system. When EDBS calls a particular CDC/APL function it actually calls an interface function which performs the task that would have been performed by the CDC function and furthermore returns to EDBS exactly what it expected when it called the CDC function.

The CDC and MTS APL file systems differ primarily in their file sharing facilities (i.e., access authorization and concurrency control). With these exceptions, simulation of the CDC/APL file system using the available MTS APL file system was a straight forward task.

The EDBS approach to concurrency control requires setting logical locks on such objects as relations, segments and records and physical locks on files. EDBS manages its own logical locks and relies on the file system tc set physical locks. One exception to this is the relational system of the APL*PLUS version of EDBS which does not rely on the file system for locking.

The conversion at University of Calgary from APL*PLUS to CDC/APL involved extensive modification of the locking mechanism used by the relational system. Modification of the hierarchical and network systems was far less extensive involving only the physical level of locking. A result of modifications made at University of Calgary was that the locking strategies used by all three systems are now similar (i.e., the relational system was modified to rely on the CDC/APL file system just as the hierarchical and network systems do).

In this section we describe the existing EDBS implementation along a number of dimensions. First, we examine data usage in general by reviewing EDBS files and authorization at the file system level. Second, we examine implementation of concurrency control in the APL*PLUS version of EDBS. Third, we look at major differences in the CDC/APL version. Finally, we consider EDBS under MTS.

1. EDBS Files and Authorization

This section describes EDBS files and access requirements at the file system level. Access restrictions are also imposed by EDBS itself. All of these files are created by various DBA utilities except for the message file, DBA log file and buffer files. (See Appendix C for a graphical representation of EDBS file usage.)

(a) Message File:

The message file contains error messages used by various EDBS utilities. There is one message file per EDBS installation. All users have read access to this file. Only the system administrator is permitted to modify this file. The message file is taken from the EDBS distribution tape.

(b) Translation Table File:

The primary purpose of the translation table is to provide a directory of all the databases administered by one DBA. There is one translation table per DBA. All users have read

access to this file. Only the DBA has complete access to this file.

(c) Record Table File:

The record table stores information about the physical aspects of relations, segment types, and record types. There is one record table per DBA. All users have read access to this file. Only the DBA has complete access to this file.

(d) Field Table File:

The field table contains information on the physical characteristics of domains, fields, and data items. There is one field table per DBA. All users have read access to this file. Only the DBA has complete access to this file.

(e) Log File:

The log file is used to keep track of who has the database open and to keep a log of database modifications. There is one log file per database. All users have read/write access to this file.

(f) Record File:

The record file is used to store record occurrences (segment occurrences or tuples), inverted lists, and locks used to synchronize sharing of the record type, segment type or relation. There is one record file per record type, segment type, or relation. All users have read/write access to this file.

(g) Buffer File:

The buffer file is used to hold data retrieved from the database. The buffer file is created in an OPEN command and destroyed in a CLOSE command. This file can only be accessed by the user who owns it.

(h) DBA Log File:

The DBA log file is used to keep track of who the DBAs are. It consists of exactly one component, a vector of DBA account numbers. There is one DBA log file per EDBS installation. All users have read/write access to this file. The DBA log file is created by the system administrator.

A database consists of n + 1 files where n is the number of different relations, segment types, or record types in the database (i.e., n record files and one log file). Access to record files is restricted via the GRANT ACCESS utility at the database level, record type level or record occurrence level. With the exception of these restrictions any user has read/write access to all database files.

All users have read/write access to the DBA log file because any common user is potentially a DBA and the procedure for creating DBAs requires that the vector of DBA account numbers be read and updated from the new DBA account. Control over authorization of DBAs is maintained as follows: Utilities for creating DBA schema tables and updating the DBA log are stored in the system administrator's library. Only if a common user has the account of the SA will he be able to load the

procedure required to install himself as a DBA. Further, the SA must permit the appropriate workspace for read to those users who are to become DBAs.

2. Concurrency control with APL*PLUS

The APL*PLUS version of EDBS makes use of three facilities to implement the hierarchical and network locking mechanism: a file interlock function, a hold list and a time out feature.

The file interlock function is provided by the APL*PLUS file system and permits a user to explicitly lock and unlock files for exclusive use. Several users requesting an interlock on a file are queued in the order of their requests. If a user does not place an interlock on a file then he will be able to access the file even though another user has an interlock on the file. In order that a user is guaranteed exclusive use of a file, all users must place an interlock on the file prior to accessing it.

A hold list is a list of all users currently having a hold on a segment occurrence of a given type. All segment occurrences of a given type are stored in one file (i.e., the record file for the segment type). A hold list is stored as the first component of each record file.

The hold list contains zero or more triples of the form (user ID, logical index, time stamp). The logical index identifies the held segment, the user ID identifies the holding user, and the time stamp is the time at which the hold was set.

When a user requests a hold on a segment, the hold list for the segment type is checked. If the segment is not currently held, then the hold is granted by adding a new triple to the hold list. If the segment is already in hold a status code is returned to indicate that a hold is not possible.

The time-out feature is used to prevent a segment occurrence from being unavailable for an excessive period of time. A hold on a segment is guaranteed to be in effect for only five minutes. If another user requests a hold on the segment after the time limit has expired then the prior hold is removed.

During the time a segment is in hold, no other user may place a hold in the same segment and thus no modification command affecting the segment may be executed. However, other users may still retrieve the held segment at any time.

During actual modification of a segment, access to the segment type is restricted to the user performing the update. This restriction is accomplished by placing an interlock on the segment type file.

Since all users must place an interlock on a file prior to accessing it for the interlock to be effective, both retrievals and modifications take place under the protection of an interlock. To avoid unnecessary delays the interlock is placed on a file only after all processing not requiring access to the file has been completed and the interlock is removed as soon as access to the file is complete. The exact lock protocol followed by EDBS is not clear from available documentation for the APL*PLUS version.

As an integrity measure EDBS also sets its own lock on a file during modifications. This lock occupies the second component of each record type file and is a two element vector. The first element of the lock vector is 0 if the record type is not locked and 1 if it is. The second element of the lock vector is 0 when no lock is in effect and the time the lock was set when the record type is locked. The lock vector is checked on every retrieval and every modification. Ordinarily, if a user does not interrupt a modification the first element will always be reset to 0 when the modification completes. However, if a modification is interrupted and is not resumed then the first element of the lock will remain set to 1 to indicate that the file is in an inconsistent state. In this event, further modifications to the record type are prevented, but retrievals are still allowed. In either case an 'INTEGRITY CHECK' message is printed each time the file is accessed until the DBA corrects the situation.[1]

The APL*PLUS version of EDBS does not rely on the file system to implement locking in the relational system. Rather, the system itself maintains for each relation a reader list and a lock vector to implement the locking mechanism.

---

[1] This paragraph is taken from documentation compiled by the Computer Systems Research Group, University of Toronto.

All tuples from the same relation are stored in one file (the record file). The reader list and lock vector for a relation are stored in the first two components of the record file. A reader list is a matrix containing user IDs for those users currently reading the relation along with the time that each read was started. A lock vector is a two element numeric vector. The first element of the lock vector is 2 if the relation is being modified, 1 if the relation is being held and 0 if the relation is neither being held nor modified. The second element of the lock vector is the time that the hold or modification was started.

The rules which must be enforced are the following:

(a) at most one user may hold a relation at one time.

(b) a relation may not be modified when another user has it in hold

(c) no other user may access a relation while one user is modifying it.

These rules are enforced via the reader list and lock vector as follows:

(a) For reading (i.e., the GET command) EDBS waits for the relation to be unlocked for modification or in hold and then enters the user number and time onto the reader list.

(b) For holds (i.e., the GET-HOLD command) EDBS waits for the relation to be unlocked for modification or unheld and then enters a 1 into the lock vector.

(c) For modifications EDBS waits for the relation to

be unlocked and unheld, writes a 2 into the lock vector and waits for the reader list to be empty.

To ensure that one user will not tie up a relation by never releasing a hold or by interrupting a read, both holds and reads are timed out after five minutes. Thus, if a user is recorded as having held a relation for more than five minutes his hold will be dropped if another user requests to hold or modify the relation. Similarily, any row of the reader list will be ignored if it is more than five minutes old.[1]

If a relation modification command is interrupted then the relation may be totally locked for any length of time. This lock is not timed out because the relation file may be in an inconsistent state. Instead, an 'INTEGRITY CHECK' error message is printed and neither holds nor modifications are allowed.[2]

3. Concurrency control with CDC/APL

The CDC/APL implementation of locking for hierarchical and network databases differs from the APL*PLUS implementation mainly in use of the interlock function (which is unavailable with the CDC/APL file system). Two features of the CDC/APL system are used to accomplish the interlock: a file tie function and an error trapping function.

---

[1] and [2] These two paragraphs are taken from documentation compiled by the Systems Research Group at the University of Toronto.

The file tie function (FTIE) takes as arguments a vector of integers and a matrix of file names. A user must tie a file prior to accessing it. When a file is tied, it is made known to the user workspace and the integer provided is associated with the file. Thereafter, the user references the file by its number rather than by its name. A file may be tied in read/modify mode or in write mode.[1]

When a user ties a file in read/modify mode he is indicating that he wishes only to read the file and that he does not mind if another user modifies the file at the same time. When a user ties a file in write mode he is indicating that he wishes to modify the file and that he does not want other users to modify the file at the same time. Any number of users may be tied to a file in read/modify mode at the same time. Only one user at a time may have a file tied in write mode. If a second user attempts to tie a file for write he receives an error message. The file system also provides a function (FUNTIE) for untieing files.

The CDC error trapping function takes as an argument a location in the program to which a branch will occur in the event of a system error. An unsuccessful tie for write may be detected (and appropriate action taken) by means of the error trapping function. The function of an interlock is achieved via the file tie and error trapping functions in the following manner:

---

[1] The CDC/APL file system also permits a file to be tied in read mode which allows other users to read (but not modify) the file at the same time. This option is not used by EDBS.

(a) A file is normally tied in read/modify mode. In fact, the OPEN command ties all database files for read/modify.

(b) Immediately prior to any write operation the file is untied and an attempt is made to tie the file in write mode.

(c) If the tie for write succeeds the write operation is performed. Otherwise, EDBS waits for one minute and tries again to tie the file for write.

(d) If a successful tie for write is not possible within five minutes EDBS gives up and returns a status code to indicate an unsuccessful lock.

(e) Immediately after performing a write operation the file is untied and tied again in read/modify mode.

The CDC/APL version of EDBS (in contrast with the APL*PLUS version) does rely on the file system to implement locking in the relational system. The approach is similar to that for the hierarchical and network systems.

The file tie and error trapping functions (not a lock vector) are used to effect locking for modification. Component one of each record file contains a hold list (not a reader list). Since a user holds the whcle relation, a hold vector consists of exactly two elements. Element 1 is the user number of the user currently holding the relation. Element 2 is the time at which the hold was started. If the relation is not in hold the hold vector is (0,0).

The following procedure is used by all three systems to avoid lost updates. (Suppose that an EDBS data manipulation command reads and writes variable X from file F):

```
Loop:  Read X from F
       Tie F for WRITE
       If tie fails go to Wait
       Write X to F
       Tie F for READ/MCDIFY
       Stop
Wait:  Delay 1 minute
       Tie for READ/MODIFY
       Go to Loop
```

This procedure amounts to back out of previous operations. The read is done and then later if it is determined that another user has the file tied for write, the read is ignored and redone. The procedure prevents lost updates because a given user's update is made only if another user was not updating the file between the given user's read and update.

There are occasions when a user must lock more than one file at a time. As an example, consider the DELETE command in the hierarchical system which triggers deletion from several record files. The CDC/APL version of EDBS handles this situation by attempting to tie for write all required record files before updating any of them. If any of the files cannot immediately be tied for write EDBS unties any that it has successfully tied, waits and then tries again to tie them all for write. Examination of EDBS APL code indicates that whenever an EDBS operation (i.e., GET, UPDATE, INSERT, DELETE, CREATEDBA, DESTROYDBA, GRANTACCESS) must modify data from more than one file, all files are tied for write before any of them are modified.

We conclude this subsection with a brief description of the CDC file access authorization facilities.

A user controls access to his files by means of a file category, password, and mode which may be specified when the file is created from APL via the FCREATE function. The following excerpts from the CDC APL reference manual (1978) describe these facilities.

(a) The file category is ordinarily private. Private files cannot be accessed by other users. A file may, alternatively, be assigned a category of semiprivate or public. Either of these categories allows other users to access the file if they know the password, the name of the file, and the user number under which it was stored.

(b) The file can be given a password. Only users who know the password can use the file.

(c) The file mode is used to control the type of operation another user can perform. For files created by APL (including workspaces) other users are ordinarily allowed to read the file (assuming the password and category do not exclude them) but are not allowed to alter or destroy the file. Other users can be given permission to alter the file by specifying the WR option (for write) when the file is created.

A file category, password, and mode may also be reassigned or changed outside the APL environment. EDBS uses the file category and mode to control access to files. The password option is not used for this purpose.

4. EDBS Under MTS

In this section we examine the MTS facilities for access authorization and concurrency control and we look into the details of simulating the corresponding CDC/APL functions via these facilities. We conclude this section with a description of the EDBS state at which our problem starts -- a state in which all CDC facilities are available via MTS functions with the exception of concurrency control which is provided via the MTS automatic locking mechanism. We say that our problem starts here because this state provides very little concurrency, incurs the risk of deadlock and jeopardizes database correctness.

The MTS/APL file system allows users to perform operations on files stored outside the APL environment. The file system consists of a set of functions which may be executed from APL or from within an APL function. To use the file system a user takes a copy of it (from a public library or the SA's account) into his active workspace. Among others, the following MTS operations are possible from within APL: CREATE, DESTROY, READ, WRITE, EMPTY, LOCK, UNLK, RENAME, and RENUMBER.

A subset of the capabilities provided by the MTS PERMIT command is available via the MTS/APL SHARE command. This command allows a user to explicitly permit other users to read from or write into his files.

The concurrency control facilities available are basically those provided by MTS which include three levels of locking, automatic locking, and explicit locking. The following excerpts from UBC Files and Devices (1976) and UBC Commands (1977) describe these facilities:

Three classes of locking are provided for maintaining integrity of a shared file: lock for reading, lock for modification, and lock for destruction. The three locking classes are inclusive in the sense that locking a file for modification also locks it for reading and locking a file for destruction also locks it for reading and modification.

The rules for concurrent use of a file among separate tasks are the following:

(a) Any number of tasks can have a file locked for reading at the same time as long as no other task has the file locked for modification or destruction.

(b) Only one task can have a file locked for modification at one time, and then only if no other task has the file locked for reading or destruction.

(c) Only one task can have a file locked for destruction at one time, and then only if no other task has the file open or locked for

reading or modification.

Files are implicitly (automatically) locked and unlocked by MTS whenever a user reguests something of MTS which requires locking. For example, if a user is operating from the subroutine level,[1] on the first call to a subroutine to read or write a line from a file, MTS will implicitly lock the file for reading (or modification) and leave the file locked in that manner until use of that file is complete. In general, the locking is associated with the FDUB (file or device usage block) associated with the file and automatic unlocking is done when the FDUB is released.

When MTS implicitly locks a file through a particular FDUB, it may raise[2] the locking class but it will never lower it. For example, if a user operating from the subroutine level first writes a line to a file and then reads a line from the same file, MTS will implicitly lock the file for modification before the first write and leave it locked for modification thereafter.

If, while attempting to lock a file MTS determines that according to the concurrent use rules it cannot lock the file as requested, MTS implicitly and automatically attempts to queue the task to wait until the file can be locked. Before doing so, however, MTS checks to ensure that queuing the job to wait on

---

[1] We distinguish between the MTS command level and the MTS subroutine level. The facilities available from within APL are those at the subroutine level.

[2] Lock for destruction is said to be at a higher level than lock for modification which is, in turn, at a higher level than lock for reading.

the file will not result in a situation wherein the current task as well as others will be deadlocked indefinitely in their respective queues. If such is the case, MTS will not allow the task to wait but instead will return an error indication.

Files can also be explicitly locked and unlocked from within APL (via the LOCK and UNLK commands). In addition, an option may be specified to request that no waiting be done if locking cannot be accomplished.

Implicit locks may be removed from within APL via the FRELEASE command which also closes the file. Otherwise, implicit locks will remain in effect until the user leaves APL.

Explicit locks may be released explicitly via the UNLK command (or the FRELEASE command which also closes the file). A special function available to EDBS (and not part of the APL file system) also permits implicit locks to be explicitly removed without closing the file.

This concludes our description of the MTS/APL file system. We now turn our attention to details of the simulation. A description of the simulated functions is available in Appendix D. We focus here on the important ones: MTSCREATE, MTSTIE, MTSUNTIE, and TRAP1.

The MTSCREATE function simulates the CDC FCREATE function. The syntax of the FCREATE function is as follows:

```
'file-name[:password] [/options]' FCREATE fnum
```

The list of options can include any of the following: DA, C, WR, S, or PU to specify direct access, coded, write mode, semiprivate or public. Since all EDBS files are direct access files and no EDBS files are coded files, these options were not simulated (i.e., files are automatically direct access and structured). Since EDBS does not use the password option, it was not simulated. MTSCREATE accomplishes the following remaining functions:

(a) An APL internal format file is created whose name is given by file-name.

(b) If the S or WR options are specified the file is permitted for read or write respectively via the MTS/APL SHARE function.

(c) The file tie number (fnum) is inserted into a table of file tie numbers and the file name is inserted in the same relative position in a table of file names.

The MTSFTIE function simulates the CDC FTIE function whose syntax is the following:

'[*account] file-name [:password] [/options]' FTIE fnum

The password option is not used by EDBS and not simulated. The list of options -- RD for read, RM for read/modify, and no option for write -- are not easy to simulate. We leave this task for further examination. MTSFTIE accomplishes the following remaining tasks:

(a) The account number if provided is concatenated to the file name to produce the MTS name account:file-name.

(b) The file is tied. That is, the file number and filename tables are updated.[1]

The MTSUNTIE function simulates the CDC FUNTIE function which takes one argument, a vector of file tie numbers. The function of MTSUNTIE is to delete the file tie numbers and corresponding file names from the file number and file name tables respectively.

The MTS TRAP1 function is used in place of the CDC error trapping function. The CDC error trapping function does not appear possible to simulate. The MTS TRAP1 function gets called from EDBS but does nothing.

Given that all facilities of the CDC/APL file system are available via the interface file system with the exception of the CDC locking and error trapping capabilities, the following problems can be expected when EDBS is run under MTS in a concurrent environment:

(a) very little concurrency

(b) loss of database correctness

(c) deadlock

Very little concurrency is possible because MTS automatic locks are not autcmatically released until the user holding the locks signs off from APL. The following examples illustrate this point for the hierarchical system:

(a) When a user GET-HOLDs a segment, the hold vector

---

[1] Each user has his own file number table and file name table which reside in his active workspace to record files currently tied and which are destroyed when he leaves APL.

in the record file must be checked and updated. When the hold vector is updated the record file is automatically locked by MTS for modification. Thereafter, other concurrent users will be prevented from retrieving segments from the record file via either the GET or GET-HOLD commands.

(b) If user A reads a segment from the record file, the file is automatically locked for read. Concurrent users are prevented from holding any segment of the same type until user A leaves APL because the record file cannot be locked for modification to update the hold vector.

(c) A DBA may not create a new database until all concurrent users who have read from the schema tables have signed off from APL. This is because the schema tables may not be locked for modification by the DBA if a common user has them locked for reading.

Within the MTS APL environment, if two users read from a file and then both attempt to write into the file they will be deadlocked each waiting for the other to release his shared lock before lock for modification can be accomplished. If deadlock is detected by MTS an error message will be passed back to EDBS but EDBS will not be able to correctly interpret it. Rather, EDBS will continue operation as if both writes had been successful. Correctness of the database is not maintained because an intended update has not been made.

C. Problem Formulation

EDBS has a logical level of locking and a physical level of locking. The logical level of locking is the GET-HOLD mechanism. Physical level locking is that done by the file system. With the CDC/APL implementation, physical locking would include the waiting and checking done by EDBS to determine if a file is tied for write.

At the logical level, an entity is a segment occurrence, record occurrence or tuple. The granule for read is a segment occurrence, record occurrence or relation. The granule for modification is a segment type, record type or relation. Logical level locking refers only to databases created by EDBS. Other data maintained by EDBS such as those contained in the schema tables are never locked at a logical level. Actions read, write, create or destroy entities. In the relational system, for example, the only actions are GET, PUT, UPDATE and DELETE.

At the physical level entities are records and the granule is a file. Locking applies to all data maintained by EDBS. An action is a file system operation for reading, writing, inserting or deleting records.

This section describes EDBS concurrency control facilities in terms of the basic concepts presented in chapter II. First, we define an EDBS transaction as our basic unit of consistency. Second, we examine the EDBS lock protocol both at the logical level and at the physical level. Third, we compare the various EDBS implementations in terms of level of consistency and amount of concurrency provided. Finally a list of key problems

associated with running EDBS in a concurrent MTS environment is provided.

The reader is advised that the MTS/APL version of EDBS referred to in this chapter is the one which is operational in an MTS environment as a single user system. That is, we assume that all file system requirements have been simulated except for concurrency control.

1. Transaction Definition

Recall that a transaction is defined as a group of actions by the same user which when executed alone transforms a consistent database into a new consistent state. The following options were considered for choosing an EDBS transaction:

(a) APL terminal session

(b) APL program containing EDBS function calls

(c) group of EDBS commands

(d) single EDBS command

Before one of the above options can be selected, the terms database, entity, action and user must be defined in the context of EDBS.

EDBS is used to create and manipulate databases which comprise the record files. In addition, EDBS maintains other data such as the schema tables and DBA log. Our concurrency problem is concerned with maintaining correctness of all EDBS files(not just the record files). Hence, we must define a database as all data maintained by EDBS.

There are three users of EDBS: SAs, DBAs and common users. Only common users or DBAs who have become common users are database users(i.e., users of databases created by EDBS). Since our concurrency problem is concerned with all EDBS data, we must define a user as an EDBS user.

There are several alternatives for defining an entity. For example, an entity could be a logical item such as a segment, record or tuple or an entity could be a physical item such as a file or file component. In addition, an entity could be something smaller than either of these items such as an index value, field value or data-item.

We select a record in a file as an entity because at our level of concern (the file system level) these are the entities.

A single record is used to store all domains, fields or data-items within a tuple, segment occurrence or record occurrence. Thus, in certain contexts we will use the term entity to refer to a tuple, segment or record occurrence (and vice versa).

Finally, we must define an action. Actions read, write, create or destroy entities. Any other computations are assumed to occur indivisibly with some action. We choose as actions the individual file operations for reading, writing, inserting, deleting records.

There are two broad types of operations supported by EDBS: Data manipulation commands such as GET, PUT, UPDATE, DELETE and other utilities such as those for creating and destroying DBAs, creating and destroying databases and granting access to databases. Execution of each EDBS operation generates a certain

sequence of actions. For example, consider the DELETE command in the hierarchical system which performs the following tasks:

(a) Check schema tables to ensure that the user has the necessary access rights.

(b) Check buffer file to ensure that the segment is being held.

(c) Lock record files for segment to be deleted and for all descendant segments.

(d) Delete segment and its descendants.

(e) Unlock record files.

The actions associated with the DELETE command include reads from the schema tables and deletions from the record files. We do not consider read/write operations on the buffer to be actions because the buffer file stores only temporary information required on a per user basis. Locking done by the file system is also not considered to be an action. Locking done by EDBS (i.e., the hold mechanism) does generate an action. An entity in the record file (the hold vector) is updated.

As another example, the EDBS utility required to create a DBA consists of operations to create the schema tables and update the DBA log file. Actions associated with CREATEDBA include creation of a number of entities in the schema tables and update of an entity in the DBA log (i.e., the vector of DBA account numbers).

EDBS does not provide a mechanism whereby actions by the same user may be grouped into consistency preserving units called transactions. (Recall the System R BEGIN-TRANS and END-TRANS operators for this purpose.)

We define an EDBS transaction as a group of EDBS operations which have been carefully chosen by a user such that the resulting sequence of actions is consistency preserving.

This is option (c) above. As noted, a problem with EDBS in its current state is that the concurrency control mechanism has no way of knowing where one transaction ends and another begins.

Given that it is possible to support option (c) for defining a transaction, it would be a minor generalization to support option (b), an APL program containing EDBS commands. The difficulties which concerned INGRES designers when considering this option are not a concern here because EDBS does not support transaction backout. Hence, additional overheads for backout through system calls is not expected.

An APL terminal session [option (a)] is not a suitable unit for defining a transaction because it would be too restrictive of concurrency. For example, if it were deemed necessary to support consistency level 3 of System R, the requirement for read reproducibility would mean that another user could not modify an entity which has been read by a given user from the time the entity was read until the given user signs off from APL.

An EDBS utility could serve by itself as a transaction because it is consistency preserving. The remainder of this subsection discusses why option (d), a single EDBS data manipulation command, could not serve as a transaction.

EDBS data manipulation commands are not in general consistency preserving. Consider the REPLACE command in the hierarchical system which updates the segment obtained by the preceding GET-HOLD command. The REPLACE command cannot be considered to be a transaction because it updates only one segment occurrence and if a consistency constraint relates two segment occurrences, this command when executed alone will not be consistency preserving.

2. Locking Protocol

Given that we are able to group actions into transactions the next step is to provide a lock protocol which guarantees that concurrent execution of transactions will result in serializable schedules.

The EDBS lock protocol (i.e., the GET-HOLD mechanism) is not sufficient to guarantee serializable schedules. The major problem is that the GET-HOLD command is restricted to operate on at most one relation, segment or record occurrence at one time. To see the problems associated with this approach consider the following example:

Suppose that exactly one consistency constraint applies to the database that "Segment S1 is equal to Segment S2". Suppose also that two EDBS transactions, T1 and T2, update segments S1 and S2 according to the following schedule:

{(T1, GET-HOLD, S1), (T1, REPLACE, S1),
(T2, GET-HOLD, S1), (T2, REPLACE, S1),
(T2, GET-HOLD, S2), (T2, REPLACE, S2),
(T1, GET-HOLD, S2), (T1, REPLACE, S2)}

As usual, we do not show buffer commands. Modification of the segment in the buffer is assumed to occur indivisibly with the REPLACE command.

The above schedule is not serializable according to Eswaran et al. (1976) because T1 updates S1 before T2 does and T2 updates S2 before T1 does. Thus the EDBS lock protocol at the logical level is unsuitable as a mechanism for controlling concurrency to maintain database correctness.

EDBS physical level locking appears to serve two purposes:

(a) to guarantee correctness of a single EDBS operation

(b) to implement part of the logical level locking

This is not an uncommon approach. Recall the System R lock mechanism in which physical locks on pages are used to ensure that operations at the Relational Storage Interface give correct results. Also, an option is available whereby physical locking may be used to accomplish logical locking.

With regard to (a) above, the only guarantee made by the CDC/APL implementation is that lost updates will be prevented. The mechanism for doing so is illustrated in the following example: (Suppose we have two transactions, T1={R1,W1} and T2={R2,W2}, such that S(R1)=S(W1)=S(R2)=S(W2)={x}.) An update will be lost with either of the following schedules:

S1={R1[X]R2[X]W1[X]W2[X]}

S2={R1[X]R2[X]W2[X]W1[X]}

Recall that these schedules cannot be serializable according to Papadimitriou (1979) because in S1, T1 is dead and

in S2, T2 is dead and there is no serial schedule in which either transaction would be dead.

CDC/EDBS does not allow either of the above schedules to occur. We assume that tie for write cccurs indivisibly with the write. In either schedule according to the lock protocol the second transaction to attempt to write will redo his read.

The serializability component of EDBS will transform the input schedules, S1 and S2, into the respective output schedules

S1(output) = {R1[X]R2[X]W1[X]R2[X]W2[X]}

S2(output) = {R1[X]R2[X]W2[X]R1[X]W1[X]}

Since the reads which are redone are ignored the effective output schedules are the serial schedules

S1(effective) = {R1[X]W1[X]R2[X]W2[X]}

S2(effective) = {R2[X]W2[X]R1[X]W1[X]}

With regard to (b) above, the CDC/APL implementation does not provide the level of isolation from other users originally intended for the system. Recall that the APL*PLUS implementation guarantees that during execution of any modification command involving a record (segment, relation) the user will have exclusive use of the record type (segment type, relation).

Exclusive use of a file, record type or otherwise, is never guaranteed with the CDC/APL implementation. Recall that the CDC/APL lcck protocol is such that when one user is writing to a file other users may read from the file. The CDC/APL implementation does ensure that during any modification command other users will be prevented from modifying the record type

(segment type, relation).

3. Level of Consistency

The System R multiple levels of consistency is concerned with the view that a transaction has of the database. Even though a certain sequence of actions is not consistency preserving, it may have a consistent view of the database.

In this subsection, the System R levels of consistency will be applied to individual EDBS operations as a means of comparing the APL*PLUS and CDC/APL versions of EDBS and to describe consistency available when EDBS relies solely on MTS automatic locking for concurrency control. It is felt that the System R notion of consistency is useful as an indicator of EDBS consistency even though we are applying this notion, not at the level of a transaction as in System R, but at the level of one EDBS operation.

Recall the System R levels of consistency:

Level 1 - least isolation from other users
- a transaction may read uncommitted data

Level 2 - a transaction reads only committed data
- read reproducibility is not guaranteed

Level 3 - complete isolation from other users
- a transaction reads only committed data
- read reproducibility is guaranteed

The APL*PLUS version of EDBS appears to provide EDBS operations with something between consistency levels 2 and 3. Read reproducibility is guaranteed, assuming that a given file is interlocked at most once within the same operation. However, an operation may read uncommitted data because file interlocks

do not appear to be held to the end of an operation.

The EDBS operations under CDC/APL achieve level 1 consistency. Since the read modify tie permits other users to write into the file at the same time, read reproducibility cannot be guaranteed. Further, an operation may read uncommitted data, again because a tie for write does not prevent other users from reading(regardless of when the tie for write is released).

What level of consistency can we expect with MTS automatic locking? Every entity read by an operation will be share locked. Every entity written will be locked for exclusive use. All locks will be held to the end of each operation.

Read reproducibility will be guaranteed because no other operation can modify an entity which is locked for read, and an operation will read only committed data because read is not possible once an entity is modified by some other user until that user releases the file containing the entity. This is level 3 consistency. Of course, the price paid for this level of consistency is that very little concurrency will be provided.

4. Amount of Concurrency

Amount of concurrency refers to the amount of interleaving of actions from different transactions.

With EDBS, interleaving of actions from different users is restricted at a logical level via the GET-HOLD mechanism. Interleaving is further restricted by physical level locking.

In this subsection, we compare the relative amounts of concurrency offered by the APL*PLUS, CDC/APL and MTS APL versions of EDBS at the level of individual EDBS operations (i.e., interleaving of actions from operat

The APL*PLUS version supports less concurrency that the CDC/APL version because the CDC/APL file system permits interleaving of reads and writes by different users to the same file, whereas the APL*PLUS implementation does not permit interleaving of any actions by different users involving the same file.

To clarify, suppose entities X, Y, and Z are stored in the same file and consider the following sequences of actions within schedules S1, S2 and S3:

S1={... R1[X]R2[Y]R1[Z] ...}

S2={... W1[X]R2[Y]W1[Z] ...}

S3={... W1[X]W2[Y]W1[Z] ...}

The CDC/APL lock protocol would permit the sequence of actions shown in schedules S1 and S2 but not S3. The APL*PLUS lock protocol would not allow the sequence of actions shown in any of these schedules.

The amount of concurrency expected with EDBS under MTS lies somewhere between that of the APL*PLUS and CDC/APL versions. MTS supports interleaving of reads(but not reads and writes) by different users involving the same file. That is, schedule S1 but not schedules S2 and S3 would be possible responses of the MTS/APL lock protocol.

5. Key Problems

This subsection identifies key problems which must be solved before EDBS is operational in a concurrent MTS environment:

(a) EDBS does not support a basic unit of consistency.

(b) The lock protocol at the logical level is not sufficient to guarantee serializable schedules.

(c) Very little concurrency is provided due to MTS automatic locking.

(d) Correctness may be destroyed if a deadlock situation should arise at the physical level.

Chapter IV: Recommendations

In this chapter we present several alternative approaches to permitting concurrent use of the Educational Data Base System. The objective of each approach is to guarantee correctness of data maintained by EDBS. We assume that BEGIN-TRANS and END-TRANS operators are available as a means of defining transactions. We conclude this chapter by recommending an approach and specifying supporting routines.

A. Alternatives

In this section we investigate three alternatives. Alternatives 1 and 2 are similar in that they both recommend relying on the file system to do logical level locking. The two alternatives differ in their proposed lock protocols. This has implications in terms of freedom from deadlock and the need for transaction backout. Alternative 3 proposes not to rely on the file system to do logical level locking. This approach would require extensive modification of EDBS and is not discussed here in detail.

Alternative: 1

The lock protocol is described as follows:

(a) A transaction has a locking phase and an unlocking phase.

(b) The locking phase is provided by the MTS automatic locking facilities (i.e., Every entity read by a

transaction will be locked for read immediately prior to the read action. Every entity written will be locked for modification immediately prior to the write action. The lock class for an entity may be raised within a transaction but it will never be lowered).

(c) All locks are automatically released as the last step in the transaction. This constitutes the unlocking phase of the transaction.

This lock protocol is not deadlock free. Transaction backout procedures are proposed for deadlock resolution.

This approach assumes that, except for deadlock, the MTS automatic lock protocol ensures serializable schedules. The proposed lock protocol appears to be a simplified version of that used by System R level 3 transactions (i.e., without intention locking).

The proposal to rely on the file system to do locking means that the actual granule locked will be one file (i.e., a segment type, record type, relation). Hence, when we say that a transaction locks an entity we mean that the transaction locks the file containing the entity.

The following example illustrates how alternative 1 might work: Suppose that each of two transactions attempt to update segments S1 and S2. Suppose also that segments S1 and S2 are of different types. The following sequence of operations will result in deadlock:

```
{(T1, GET-HOLD, S1),   (T1, REPLACE, S1),
 (T2, GET-HOLD, S2),   (T2, REPLACE, S2),
 (T1, GET-HOLD, S2),   (T2, GET-HOLD, S1)}
```

T1 will be waiting to lock the record file to read the hold vector of segment type S2. Similarly, T2 will be waiting to lock the record file to read the hold vector for segment type S1. Neither transaction will be able to lock its respective record file because the other transaction is holding an exclusive lock on the file, obtained during the previous GET-HOLD operation.

MTS will detect this deadlock situation and pass a message back to one of the transactions indicating that the read (of the hold vector) has been unsuccessful. This deadlock message could be a signal for transaction backout. In our example, if T1 receives the deadlock message then the prior update of segment S1 by T1 should be undone.

The remainder of this subsection discusses implementation considerations for transaction backout:

EDBS maintains a database log containing a list of all database modifications. A record of the user who performed the modification is not currently maintained. The existing logging procedure could be easily modified to record this information. Also, a record of the time at which the modification was started would be useful.

Transaction backout would involve undoing database modifications as recorded in the log. The BEGIN-TRANS operator could record the time at which the transaction was started as a global variable stored in the user's active work space. If it becomes necessary to back the transaction out, database modifications recorded in the log belonging to this user could be undone so long as the time at which the modification was made

is later than the time at which the transaction was started.

The System R principle of isolated backout will be necessary. Hence, the lock protocol must be such that data modified by a given transaction is not modified by any other until the given transaction terminates. If locks are released as part of the END-TRANS operator, isolated backout will be guaranteed.

A log of modifications made to EDBS data, other than that contained in the record files, is not currently maintained by EDBS (i.e., modifications to schema tables and the DBA log). These modifications could be logged and DBA utilities could then be handled like any other transactions (bracketed by BEGIN-TRANS and END-TRANS operators).

Alternative: 2

The lock protocol is described as follows:

(a) All entities required by a transaction are automatically locked in the required mode in one step at the beginning of the transaction.

(b) If the locking mode is to be raised within the transaction (i.e., from read mode to write mode) then the highest level locking mode is requested.

(c) All locks are automatically released at the end of the transaction.

This alternative has the advantage that transaction backout procedures for deadlock resolution are not required.

The lock protocol is virtually identical to that proposed by INGRES differing only in its implementation details. Recall that INGRES avoids deadlock by requiring that an interaction locks all required entities before proceeding.

The locking phase of the lock protocol could be implemented via MTS/APL explicit lock facilities as part of the BEGIN-TRANS operator. Similarly, the END-TRANS operator could automatically release all locks held by the transaction.

As with Alternative 1 the proposal is to rely on the file system to do all locking. Hence, the actual granule locked will be one file.

The event in which a transaction cannot lock all required entities will be realized in the form of deadlock. If lock requests deadlock, then the transaction which receives the MTS deadlock message could return a status code to the user to indicate that he should attempt to run his transaction at a later time and all locks currently held by the transaction could be released. Since the transaction has not yet executed any actions, transaction backout is not required.

Alternative 2 would be practically impossible to implement in an EDBS context because it would be difficult to determine locking requirements at the beginning of a transaction. Consider the following problems:

(a) The whole transaction would have to be analyzed prior to execution to determine which files will be required.

(b) Within a single EDBS data manipulation command or utility, more than one file may be required. There is no way to determine the files required by examining the name of the command or utility.

(c) Similarly, the mode of lock cannot be determined from knowledge of the command or utility.

(d) It is often impossible to determine either the desired mode of lock or the files required because these depend on the results of previous actions.

A solution to these problems might be to assume the worst. Hence, for all transactions composed by common users we could assume that read access to the schema tables is required. Further, we could assume that every transaction requires write access to all database files. Hence, the n record files and the database log file could be locked for modification and the schema tables could be locked for read at the beginning of each transaction.

DBA utilities do not require access to database files. A BEGIN-TRANS operator specific for DBAs could lock the DBA log file and the schema tables for exclusive use.

Both alternatives 1 and 2 are restrictive of concurrency because locks are held on files rather than on individual entities. Alternative 2 is more restrictive than Alternative 1 due to the requirement that all locks must be requested at the beginning of each transaction. The amount of concurrency provided by Alternative 2 will be further restricted by implementation requirements if we choose to make worst case

assumptions about the files required by a transaction. A major advantage of alternative 2 over alternative 1 is that transaction backout procedures for deadlock resolution are not required.

Alternative: 3

Alternative 3 differs from previous approaches in that we do not rely on the file system to do logical level locking. Further, we provide the user with locking facilities which he will use as part of a lock protocol to ensure database correctness. We do not discuss here the special facilities which a DBA may need. We focus only on locking requirements for a common user.

Alternative 3 proposes that the user be provided with the capability of holding more than one logical item at one time. For simplicity, a lock protocol could be enforced which states that a user must hold all logical items which he intends to modify at the beginning of his transaction. Consistent with the existing approach, a status code could be returned to the user to indicate an unsuccessful hold if any of the items requested cannot be immediately placed in hold. This very simple lock protocol avoids deadlock and hence deadlock detection and transaction backout procedures are not required.

The facilities proposed are basically those of the IMS GET-HOLD mechanism. Recall that IMS does not enforce the restriction (as does EDBS) that a user may have at most one logical entity in hold at one time. Since IMS does not enforce the simple lock protocol which we have proposed, user programs may deadlock, and backout procedures are provided.

When a logical item (i.e., segment, record, relation) is actually being modified in the database it must be locked for modification. With the existing approach, EDBS manages its own locks for read (i.e., holds) and relies on the file system to lock for modification.

Alternative 3 delegates additional responsibility to EDBS for lock management. A lock for each segment, record or relation in the database is already maintained in the form of a hold list. A code could be added to each (user ID, logical index, time stamp) triple in the hold list to distinguish between a read lock and a write lock. Additional procedures will be required to set and release locks for modification.

Alternative 3 will require that MTS automatic locks be released at some appropriate time. MTS locks could be released immediately after the action responsible for setting them or, alternatively, automatic locking could remain in effect for the duration of a single EDBS operation. Certain EDBS storage operations may initiate procedures to reorganize inverted list blocks. This provides at least one motivation for maintaining file locks for the duration of each individual EDBS operation. Recall that System R physical level locking was motivated by similar requirements.

Alternative 3 offers a greater amount of concurrency than alternatives 1 and 2 for the following reasons:

(1) logical items (i.e., a segment) may be locked rather than requiring that an entire file (i.e., the segment type) be locked.

(2) The user has information about his transaction which he can incorporate into his lock protocol to increase concurrency. For example, a user knows when he is finished with an entity and he may explicitly RELEASE it for use by other users.

A major disadvantage of alternative 3 is that database correctness is not guaranteed unless all users employ an appropriate lock protocol.

In this section we have proposed three alternatives which may be summarized as follows: Alternatives 1 and 2 make two proposals in common: First, to rely on the file system to do both logical and physical level locking. Second, to automatically lock all entities required by a transaction thus freeing the user of the responsibility for any particular locking protocol. Alternative 1 recommends relying on MTS automatic locking for the lock phase of its lock protocol. The lock protocol admits deadlock and transaction backout procedures are proposed to resolve it. Alternative 2 recommends that entities required by a transaction be automatically locked via MTS explicit locking at the beginning of a transaction. The lock protocol is such that if deadlock occurs, any transaction involved in the deadlock has not yet executed any actions and hence transaction backout procedures are not required.

Alternative 3 proposes to provide additional facilities for locking logical items and to leave most of the responsibility for lock protocols up to the user. Dependence upon the file system to do logical level locking is contra-indicated.

B. Selected Approach

Alternative 2 (i.e., automatic locking of all entities at the beginning of a transaction and release of all locks at the end of a transaction) is chosen largely due to its implementation simplicity.

Two types of BEGIN-TRANS and END-TRANS operators are proposed. These operators will not be stored as part of the interface file system. Rather, the work spaces intended for common users will contain BEGIN-TRANS and END-TRANS operators for common users. Similarly, work spaces intended for DBAs will contain BEGIN-TRANS and END-TRANS operators for DBAs. The SA is not provided with transaction definition facilities because his function is to set up the system initially and to modify the system if required. We do not expect him to be using EDBS concurrently with DBAs and common users.

The proposed solution makes use of the following information about EDBS:

(1) Common users do not modify information in the schema tables.

(2) EDBS data manipulation commands do not require access to the DBA log file.

(3) DBA utilities do not require access to database files.

(4) Database files (i.e., record files and database log) remain tied when EDBS returns control to the user so long as the database is open.

The remainder of this section provides specifications for the BEGIN-TRANS and END-TRANS operators and outlines details of the implementation.

(1) DBA Operators

BEGIN-TRANS
- lock the schema tables for destruction
- lock the DBA log file for modification

END-TRANS
- release all locks held on schema tables and DBA log file

(2) Operators for Common Users

BEGIN-TRANS
- lock all record files and the log file (belonging to the currently open database) for modification
- lock the schema tables for read

END-TRANS
- release all locks held on database files and tables

(3) Implementation Details

(a) The BEGIN-TRANS operator for DBAs must lock the schema tables (ZTTABLE, ZRTABLE, ZFTABLE) for destruction. This lock class is chosen rather than modification because at least one of the DBA utilities (DESTROYDBA) requires that the schema tables be locked for destruction. The higher lock mode is not expected to inhibit concurrency because the DBA is the only user who would require write access to the schema tables and

lock for destruction by a DBA would not exclude him from also writing. Common users who require read access to the schema tables will be excluded by either locking class.

(b) The BEGIN-TRANS operator for DBAs must lock the DBA log file for modification. The SA's account number must be known to accomplish the lock. We recommend storing the system dependent variable SYSACCNT in all DBA work spaces.

(c) The BEGIN-TRANS operator for common users must lock the schema tables for read and database files for write. Names of database files are available from the interface file system variable FNAMES. Only filenames from FNAMES whose corresponding tie number from FNUMS is greater than 15 should be selected as database files. Locking the schema tables will require the DBA account number.

(d) The END-TRANS operator for common users must unlock the schema tables and database files. The procedure for locating file names outlined for the BEGIN-TRANS operator could also be used here.

(e) We recommend that files be locked via the FMTS LOCK function, and that they be unlocked via the FRELEASE function. Recall that FRELEASE also closes the file. Investigations into advantages of unlocking files without closing them has led us to conclude that there are none (i.e.,

reductions in overhead by not closing files would be insignificant).

In this chapter we have described three alternatives for providing concurrent use of the Educational Data Base System, and we have selected an approach based on feasibility of implementation. We conclude this chapter with a summary of our recommendations:

(1) BEGIN-TRANS and END-TRANS operators should be provided by means of which a user may define his own transactions.

(2) Entities required by a transaction should be automatically locked as part of the BEGIN-TRANS operator and automatically unlocked as part of the END-TRANS operator.

(3) EDBS should rely on the file system to do all locking required by its concurrency control mechanism.

(4) In terms of implementing (2) above, worst case assumptions should be made concerning entities required by a transaction.

The amount of concurrency provided by our recommended approach can be summed up as follows: Any number of transactions may read from the schema tables at the same time. Transactions may execute concurrently on different databases created and maintained by EDBS. Otherwise, transactions are executed in a serial fashion.

Chapter V: Conclusion

This thesis has described our experience with the EDBS undertaking and proposed a solution for running EDBS in a concurrent EDBS environment.

In chapter 2, we clarified the basic objective of concurrency control which is to ensure correctness. We then examined one approach for obtaining correctness which is to ensure that actions from transactions are intermingled in a serializable fashion. Next, we examined two lock protocols which have been shown to guarantee serializability. Finally, we reviewed concurrency control facilities provided by a number of systems.

In chapter 3, we examined EDBS concurrency control facilities and compared them to those of other systems. We provided various analyses of the existing lock protocol including an indication of the amount of concurrency and level of consistency provided. Finally, we compiled a list of key problems requiring resolution before EDBS is operational in a concurrent MTS environment.

In chapter 4, we proposed various alternative solutions which solved the key problems and we recommended an approach based on practical considerations.

The remainder of this chapter identifies areas requiring further work.

1. In this thesis we have restricted our attention to read and write actions. Those that create and destroy their entities have not been considered in detail. The notion of

serializability must be extended to account for actions which create and destroy entities. The analyses presented in chapter 2 and the alternatives proposed in chapter 3 should be examined in terms of this broader notion of serializability.

2. We have identified two types of EDBS entities: those at a logical level such as segments and those at a physical level such as records. However, some EDBS transactions access entities which may not belong to either of these two categories. For example, when a DBA creates a database he creates the record files and the database log file. If he does not also write into these files, then he has created no entities. This difficulty indicates that additional synchronization may be required for transactions which create and destroy such items as the schema tables and databases. For example, it is not clear how the proposed solution would handle the situation in which a DBA attempts to destroy a database while some user has it open.

3. The problem of phantoms has not been thoroughly investigated in the context of EDBS. If a transaction locks a set of entities as given by some predicate then other transactions must be prevented from adding an entity to this set until the given transaction terminates. Otherwise, if the given transaction makes a second retrieval based on the same predicate, a different set of entities will be retrieved. Thus, the non-existence of an entiity must be locked. This problem refers to the relational system where sets of entities (i.e., tuples) are retrieved and it also refers to

the hierarchical and network systems where only one entiity at a time is retrieved. For example, in the hierarchical system, the non-existence of a particular segment must be locked until the end of a transaction so that the transaction does not find that the segment did not exist upon a first retrieval and did exist upon a second retrieval. The alternatives presented in chapter 3 should be evaluated in terms of how they resolve phantom problems.

4. EDBS might provide a useful environment in which to investigate the relationship between the lost update problem and the problem of ensuring consistency. Papadimitriou's transaction model assumes that values for variables in a transaction's read set are read instantaneously and similarily that values for variables in the write set are written instantaneously. Given these assumptions, the only way that a history cannot be serializable is if updates get lost. EDBS avoids lost updates at a logical level. Given that EDBS supports the concept of a transaction, it might be possible to vary the lock protocol at a physical level to simulate various conditions for consistency, all the while being assured that the lost update problem is under control.

REFERENCES

Astrahan, M.M., D.D. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, S.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson. 1976. System R: Relational approach to database management. ACM Transactions on Database Systems 1(2): 97-137.

Bayer, R. and E. McCreight. 1972. Organization and maintenance of large ordered indices. Acta Informatica 1: 173-189.

Bernstein, P.A., D.W. Shipman, J.B. Rothnie, and N. Goodman. 1977. The concurrency control mechanism of SDD-1: A system for distributed databases (the general case). Computer Corporation of America TR CCA-77-09. Cambridge, Mass.

Bernstein, P.A., D.W. Shipman, and J.B. Rothnie. 1980. Concurrency control in a system for distributed database (SDD-1). ACM Transactions on Database Systems 5(1): 18-51.

Bernstein, P.A. and D.W. Shipman. 1980. The correctness of concurrency control mechanisms in a system for distributed databases (SDD-1). ACM Transactions on Database Systems 5(1): 52-68.

Bjork, L.A. 1973. Recovery scenario for a DB/DC system. Proc. ACM Annual Conf. Atlanta, Ga: 142-146.

Chamberlin, D.D. and R.F. Boyce. 1974. SEQUEL: A structured English query language. Proc. ACM SIGFIDET Workshop. Ann Arbor, Michigan. May: 249-264.

CODASYL. 1971. Data Base Task Group Report. CODASYL, DBTG, ACM, New York. April.

CODASYL Programming Language Committee. 1976. COBOL Journal of Development.

Control Data Corporation. 1978 APL Version 2 Reference Manual 60454000.

Date, C.J. 1977. An introduction to database systems. Second edition. Addison-Wesley Publishing Company. ISBN 0-201-14456-5.

Davies, C.T. 1973. Recovery semantics for a DB/DC system. Proc. ACM Annual Conf. Atlanta, Ga: 136-141.

Engles, R.W. 1971. An analysis of the April 1971 DBTG Report - a position paper presented to the Programming Language Committee by the IBM Representative to the Data Base Task

Group. (Available from British Computer Society.)

Engles, R.W. 1976. Currency and concurrency in the COBOL Data Base Facility. Proc. IFIP Working Conference on Modelling in Data Base Management Systems, January.

Eswaran, K.D., J.N. Gray, R.A. Lorie, and I.L. Traiger. 1976. The notions of consistency and predicate locks in a database system. Communications of the ACM 19(11): 624-633.

Gray, J.N., R.A. Lorie, G.R. Putzolu, and I.L. Traiger. 1976. Granularity of locks and degrees of consistency in a shared database. Pp. 695-723 in Proc. IFIP Working Conference on Modelling in Data Base Management Systems. January. Freudenstadt, Germany.

Hawley, D.A., J.S. Knowles, and E.E. Tozer. 1975. Database consistency and the CODASYL DBTG proposals. Computer Journal. 18(1): 206-212.

IBM Corporation. Information management system/virtual storage general information manual. GH20-1260.

Kedem, Z. and A. Silberschatz. 1979. Controlling concurrency using locking protocols (preliminary report). Proc. IEEE February: 274-285.

Kwong, Y.S. and D. Wood. 1979. Concurrency in B-trees, S-trees and T-trees. Computer Science Technical Report 79-CS-17, McMaster University.

Kwong, Y.S. and D. Wood. 1979. Approaches to concurrency in B-trees. (To appear in Proc. Mathematical Foundation of Computer Sciences, Rydzyna, Poland.)

Kung, H.T. and C.H. Papadimitriou. 1979. An optimality theory of concurrency control for databases. Proc. 1979 SIGMOD Conf. Boston, Mass. May.

Lien, Y.E. and P.J. Weinberger. 1978. Consistency, concurrency, and crash recovery. ACM-SIGMOD Conference.

Papadimitriou, C.H. 1977. The serializability of concurrent database updates. Journal of the ACM 26(4): 631-653.

Ries, D.R. and M. Stonebraker. 1977. Effects of locking granularity in a database management system. ACM Transactions on Database Systems 2(3): 233-246.

Ries, D.R. and M. Stonebraker. 1979. Locking granularity revisited. ACM Transactions on Database Systems 4(2): 210-227.

Rothnie, J.B. and N. Goodman. 1977. An overview of the preliminary design of SDD-1: a system of distributed

databases. Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks. Berkeley, California. May.

Rothnie, J.B., P.A. Bernstein, S. Fox, N. Goodman, M. Hammer, T.A. Landers, C. Reeeve, D.W. Shipman, and E. Wong. 1980. Introduction to a system for distributed databases (SDD-1). ACM Transactions on Database Systems 5(1): 1-17.

Schlageter, G. 1978. Process synchronization in database systems. ACM Transactions on Database Systems 3(3): 248-271.

Silberschatz, A. And Z. Kedem. 1978. Consistency in hierarchical database systems. (Available in JACM.)

Simon Fraser University. 1978. MTS/APL file system user's guide. Computing Center.

Stonebraker, M., E. Wong, P. Kreps, and G. Held. 1976. The design and implementation of INGRES. ACM Transactions on Database Systems 1(3): 189-222.

University of British Columbia 1976. UBC files and devices. Computing Centre.

University of British Columbia 1977. UBC Commands. Computing Centre.

University of Calgary. Educational data base system data base administrator's manual. Department of Computer Services CSRM-009.

University of Toronto. 1975. Educational data base system data manipulation facility user's manual. Ccmputer Systems Research Group. October.

University of Toronto. 1975. Educational data base system data base administrator's manual. Computer Systems Research Group. December.

Yannakakis, M., C.H. Papadimitriou, and H.T. Kung. 1979. Locking policies: safety and freedom from deadlock. Proc. IEEE. February: 286-297.

APPENDIX A: EDBS OVERVIEW

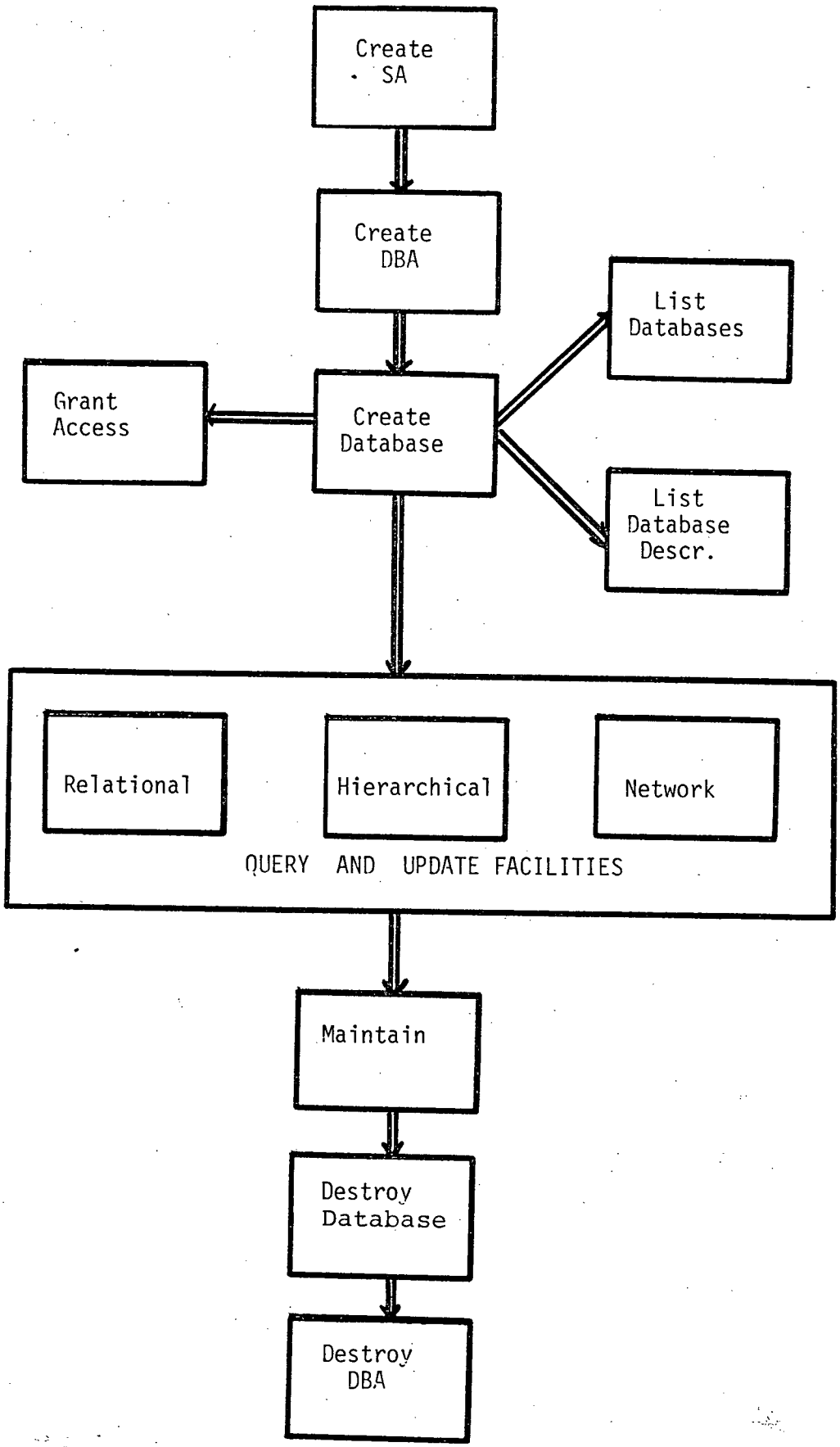

Figure 3

APPENDIX B: EDBS Commands and Utilities

Relational System	Hierarchical System	Network System
GET	GET UNIQUE	GET RECORD
GET HOLD	GET NEXT	GET SET
PUT	GET NEXT WITHIN PARENT	GET HOLD
UPDATE	GET HOLD	STORE
DELETE	INSERT	DELETE
	REPLACE	INCLUDE
	DELETE	REMOVE
		MODIFY
		CHANGE TO CURRENT

DBA Utilities

CREATEDBA	DUMPLOG
DESTROYDBA	ERASELOG
LISTDATABASES	MAINTAIN
CREATE	MAKEAVAILABLE
DESTROY	MAKUNAVAILABLE
GRANTACCESS	LISTACTIVEUSERS

APPENDIX C : EDBS FILE USAGE

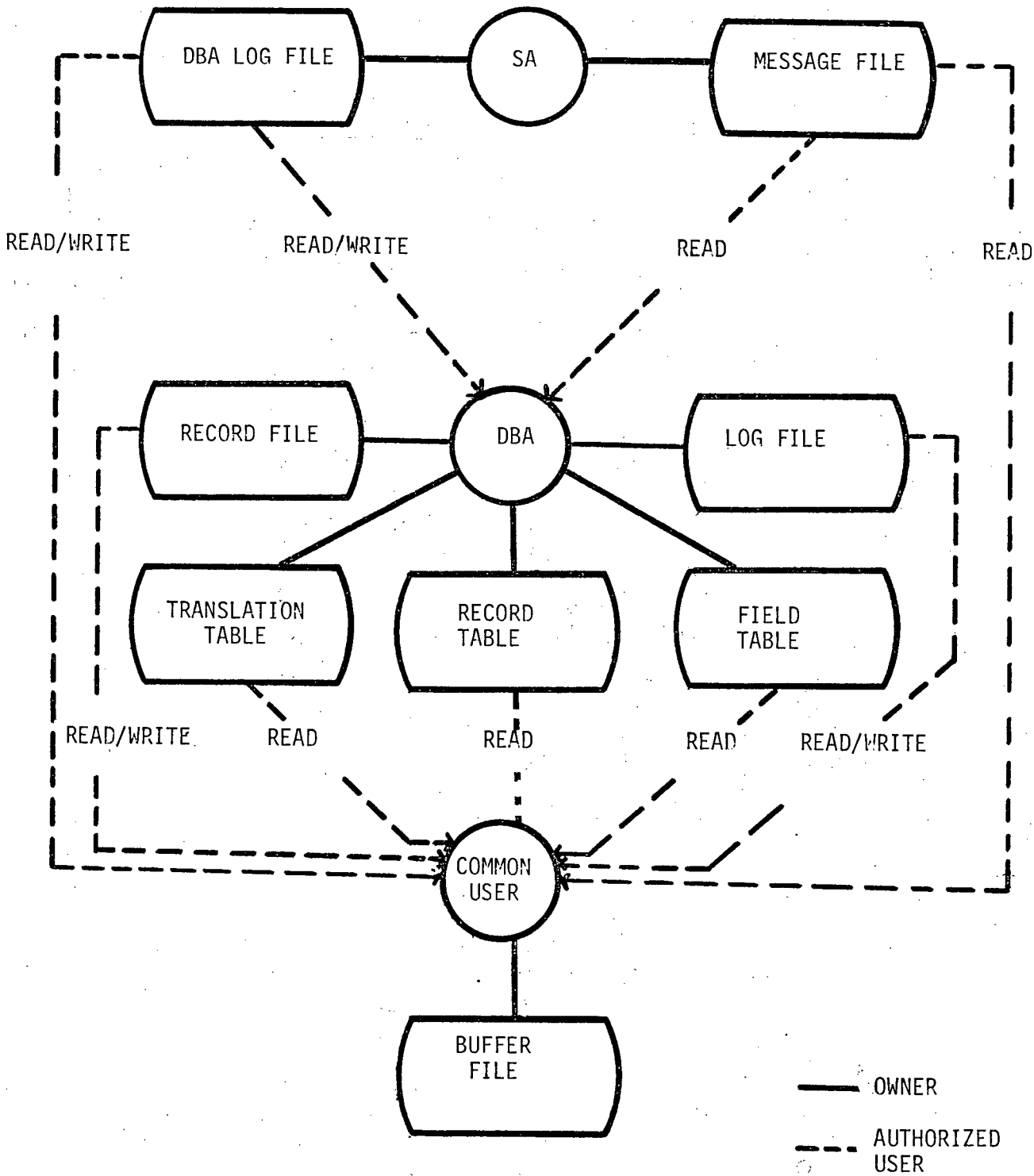

Figure 4

APPENDIX D: Interface File System Description

This appendix provides a description of the major functions which comprise our interface file system. There are basically two categories of functions which had to be simulated. First, the CDC/APL file system functions. Second, a number of system functions. Of particular importance in the later category is the #FI function. In the following, each simulated file system function is described in detail. The simulated system functions are only briefly described. The reader is referred to the CDC APL Version 2 Reference Manual for further details. Our description of the simulated file system functions is in part a restatement of pages 10.8 to 10.10 of the CDC Manual.

MTSCREATE: 'file-name [/options]'MTSCREATE fnum

The MTSCREATE function simulates the CDC file create function (FCREATE). MTSCREATE may be used to create a file and specify options about the type of file. When the file is created it is tied to the file number fnum. The list of options may include S or WR to permit the file for read or read/write, respectively, to other users. Any other options specified in the list of options are ignored. Files created by the MTSCREATE function are always internal I/O format MTS line files. Examples of file creation follow:

```
'FILE1' MTSCREATE 11 (A file named FILE 1 with 11
                      as its number)

'FILE2/DA S WR' MTSCREATE 2 (A file named FILE 2
                             permitted for RW to
                             others)
```

MTSDEL: MTSDEL fnum [, rnum]

The MTSDEL function simulates the CDC file record delete function (FRDEL). MTSDEL deletes the record rnum from file fnum. If the record was already absent, nothing is done (except that the file position changes) and nc errors result.

MTSERASE: MTSERASE fnums

The action of the CDC file erase function (FERASE) when applied to CDC direct access files is simulated by the MTSERASE function. All files specified by the right argument are destroyed.

MTSFNAMES: result <--- MTSFNAMES

The MTSFNAMES function returns a matrix of names (and user I.D.s) of files currently tied. MTSFNAMES simulates the CDC FNAMES function. The number of columns in the matrix returned is always 13 (i.e., three less than with CDC because MTS user I.D.s are three characters shorter than CDC user I.D.s). An example follows:

```
      MTSFNAMES
SAMPLE 1
ALGEBRA
*XXAR FILE1
```

MTSFNUMS: result <--- MTSFNUMS

The MTSFNUMS function returns a vector of numbers in use for tied files. The order is the same as the order of file names in the result from MTSFNAMES. MTSNUMS simulates the CDC FNUMS function.

MTSFTIE: '[*account] file-name[/options]' FTIE fnum

The MTSFTIE function simulates the CDC file tie function (FTIE). MTSFTIE gives the number fnum to the previously stored file having the indicated name. If a user account (I.D.) is given, the stored file is sought under than I.D. rather than the one used for signing onto the system. The list of options if provided is ignored by the MTSFTIE function. Examples using MTSFTIE follow:

```
'FILE5' MTSFTIE 7          (A user ties one
                            of his own files)

'*XXAR FILE1' MTSFTIE 8    (A user ties a file
                            belonging to
                            another user)
```

MTSFUNTIE: MTSFUNTIE fnums

MTSFUNTIE simulates the CDC FUNTIE. All files for which their file numbers appear in the vectcr or scaler right argument are untied. To untie all tied files, use MTSUNTIE MTSFNUMS.

MTSREAD: result <--- FREAD fnum [, rnum]

The MTSREAD function reads from the file having fnum as its file number that record having rnum as its record number. If rnum is not provided, the current record number is used. If that record does not exist, an empty numeric vector is returned. MTSREAD simulates the CDC FREAD function.

MTSSTAT: result <--- FSTATUS fnum

MTSSTAT simulates a very small subset of the capabilities provided by the CDC file status function (FSTATUS). MTSSTAT returns the largest record number currently in use by the file having fnum as its file number. If the file is empty -1 is returned.

MTSWRITE: array MTSWRITE fnum [, rnum]

The MTSWRITE function writes its left argument on the file having fnum as its number as the record having rnum as its record number. If rnum is not provided, the current record number is used. MTSWRITE simulates the CDC FWRITE function.

The $"FI functions ($"FI1, $"FI2, $"FI3 and $"FI4) accomplish the interface between EDBS and the functions described above. The following describes how they work: The CDC file system functions use the system function #FI to perform all file operations. For example, FCREATE is defined as follows:

```
"A FCREATE B <1> A #FI  1,B"
```

The number(1) following the #FI indicates that #FI should perform a file create. Each file system function calls #FI with a unique number as the first element in the right argument. The #FI function can actually be used directly and is used directly by EDBS.

The $"FI functions simulate the #FI function. All references to #FI made by EDBS have been modified to refer to the appropriate $"FI function.

Our interface file system includes a number of other functions. $"TRAP1 replaces the CDC error trapping function, #TRAP. $"LIB1 simulates the CDC system function #LIB. The MTSID and reverse MTSID (REVMTSID) functions replace EDBS facilities for decoding and encoding a user account number.

APPENDIX E: Interface File System Functions

FUNCTION NAMES :

MTSCREATE	MTSDEL	MTSERASE	MTSFNAMES	MTSFNUMS
MTSFTIE	MTSFUNTIE	MTSID	MTSREAD	MTSSTAT
MTSUNTIE	MTSWRITE	$"FI1	$"FI2	$"FI3
$"FI4	$"LIBDELETE	$"LIBUPDATE		$"LIB1
$"REVMTSID	$"TRAP1			

VARIABLE NAMES :

FNAMES FNUMS

```
    " A MTSCREATE B;FNAME;MSG;OPTIONS
<1>   FNAME=(1+A$.'%')$TAA
<2>  $*@@@ CREATE FILE CALLED FNAME @@@
<3>   $>(0$EQ$EX3$TAMSG=FMTS 'CREATE ',FNAME)%CREATED
<4>   MSG
<5>   $>0
<6>  CREATED:OPTIONS=(($,FNAME)+1)$DRA
<7>  $*@@@ UPDATE FILENAME LIST @@@
<8>   $"LIBUPDATE FNAME
<9>  $*@@@ IF SEMIPRIVATE PERMIT FOR READ TO OTHERS @@@
<10>  $>(($,OPTIONS)$LTOPTIONS$.'S')%UPDATE
<11>  $>(0$EQ$EX3$TAMSG=FMTS 'SHARE ',FNAME)%SHARED
<12>  MSG
<13>  $>0
<14> $*@@@ IF WRITE OPTION PERMIT FOR WRITE TO OTHERS @@@
<15> SHARED:$>(($,OPTIONS)$LTOPTIONS$.'W')%UPDATE
<16>  $>(0$EQ$EX3$TAMSG=FMTS 'SHARE ',FNAME,' RW')%UPDATE
<17>  MSG
<18>  $>0
<19> $*@@@ UPDATE FNAMES AND FNUMS @@@
<20> UPDATE:$>(($,FNAMES)$EQ0)%FIRST
<21>  FNAMES=FNAMES,<1> FNAME,(16-$,FNAME)$,' '
<22>  $>TIE
<23> FIRST:FNAMES= 1 16 $,FNAME,(16-$,FNAME)$,' '
<24> TIE:FNUMS=FNUMS,B
    "
```

```
    " MTSDEL B;FNUM;RNUM;MSG;FNAME;M;DATA
<1>   FNUM=1$TAB
<2>   FNAME=,FNAMES<(FNUMS$EQFNUM)%$.$,FNUMS;>
<3>  $*@@@ SET APL EXTERNAL FORMAT @@@
<4>   MSG='EXTERNAL' FSETTYPE FNAME
<5>  $*@@@ SET DIRECT ACCESS MODIFIER @@@
<6>   #IO=0
<7>   M=32$,0
<8>   M<30>=1
<9>   MSG=M FSETMODS FNAME
<10>  #IO=1
<11>  $>(($,RNUM=1$DRB)$EQ0)%SEQ
<12> $*@@@ FIND LINE NO. FOR DIRECT ACCESS WRITE @@@
<13>  MSG=(RNUM*1000) FSETLINE FNAME
<14> SEQ:$>(0$EQ$EX3$TAMSG='' FWRITE FNAME)%RESET
<15>  MSG
<16>  $>0
<17> RESET:
<18> $*@@@ INCREMENT LINE POINTER @@@
<19>  MSG='RNUM' FGETLINE FNAME
<20>  MSG=(RNUM+1000) FSETLINE FNAME
    "

    " MTSERASE A;MSG;FNUM;FNAME;I
<1>   I=,1
<2>   $>(I$GT$,,A)%0
<3>   FNUM=1$TAI$TAA
<4>   FNAME=,FNAMES<(FNUMS$EQFNUM)%$.$,FNUMS;>
<5>  $*@@@ UNTIE FILES GIVEN BY A @@@
<6>   MTSUNTIE FNUM
<7>  $*@@@ DESTROY FILES GIVEN BY A @@@
<8>   $>(0$EQ$EX3$TAMSG=FMTS 'DESTROY ',FNAME)%DONE
<9>   MSG
<10>  $>0
<11> DONE:I=I+1
<12> $*@@@ UPDATE FILENAME LIST @@@
<13>  $"LIBDELETE FNAME
<14>  $>2
    "

    " R=MTSFNAMES;I
<1>   I=,1
<2>   R= 0 13 $,' '
<3>  NEXT:$>(I$GT$,FNUMS)%0
<4>   $>(FNAMES<I;5>$EQ':')%ACC
<5>  $*@@@ COMPOSE FNAME IF NO ACCOUNT NO. GIVEN @@@
<6>   R=R,<1> ' ',4$DR,FNAMES<I;>
<7>   $>OUT
<8>  $*@@@ COMPOSE FNAME IF ACCOUNT NO. GIVEN @@@
<9>  ACC:R=R,<1> '@',(,FNAMES<I;$.4>),' ',7$,5$DR,FNAMES<I;>
<10> OUT:I=I+1
<11>  $>NEXT
    "
```

```
    " RESULT=MTSFNUMS
<1>   RESULT=FNUMS
    "

    " A MTSFTIE B;MSG;FNAME;STATUS
<1>  $*@@@ COMPOSE FNAME IF ACCOUNT NO. GIVEN @@@
<2>   $>(A<1>$NE'@')%NOACC
<3>   FNAME=A<2 3 4 5>,':',(((6$DRA)$NE' ')$.1)$DR5$DRA
<4>   FNAME=(1+FNAME$.'%')$TAFNAME
<5>  $*@@@ COMPOSE FNAME IF NO ACCOUNT NO. GIVEN @@@
<6>   $>ERRCHK
<7>  NOACC:FNAME=(1+A$.'%')$TAA
<8>  ERRCHK:$>(B$EPFNUMS)%E1
<9>  $*@@@ CHECK THAT FILE PERMITTED @@@
<10>  $>(0$EQ$EX3$TAMSG=FMTS('CHKACC ',FNAME) INTO 'STATUS')%OK
<11>  MSG
<12>  $>0
<13> OK:$>(STATUS$EQ0)%E2
<14> $*@@@ UPDATE FNUMS AND FNAMES @@@
<15>  $>(($,FNAMES)$EQ0)%FIRST
<16>  FNAMES=FNAMES,<1> FNAME,(16-$,FNAME)$,' '
<17>  $>TIE
<18> FIRST:FNAMES= 1 16 $,FNAME,(16-$,FNAME)$,' '
<19> TIE:FNUMS=FNUMS,B
<20>  $>0
<21> E1:'TIE NUMBER IN USE'
<22>  $>0
<23> E2:'FILE NOT PERMITTED'
    "

    " MTSFUNTIE A;I
<1>  $*@@@ UNTIE FILES GIVEN BY A @@@
<2>   I=,1
<3>   $>(I$GT$,,A)%0
<4>   MTSUNTIE 1$TAI$TAA
<5>   I=I+1
<6>   $>3
    "

    " Z=MTSID
<1>  $*@@@ THIS FUNCTION FINDS THE USER ID @@@
<2>  $*@@@ FOR THE CALLING ACCOUNT         @@@
<3>   Z=,#AV<#IO+(4$,256)$EN1$TA#AI>
    "
```

```
    " RESULT=MTSREAD B;FNUM;RNUM;MSG;FNAME;M;DATA
<1>   FNUM=1$TAB
<2>   FNAME=,FNAMES<(FNUMS$EQFNUM)%$.$,FNUMS;>
<3>  $*@@@ SET APL INTERNAL FORMAT @@@
<4>   MSG='INTERNAL' FSETTYPE FNAME
<5>  $*@@@ SET DIRECT ACCESS MODIFIER @@@
<6>   #IO=0
<7>   M=32$,0
<8>   M<30>=1
<9>   MSG=M FSETMODS FNAME
<10>  #IO=1
<11>  $>((($,RNUM=1$DRB)$EQ0)%SEQ
<12> $*@@@ FIND LINE NO. FOR DIRECT ACCESS READ @@@
<13>  MSG=(RNUM*1000) FSETLINE FNAME
<14> SEQ:$>(0$EQ$EX3$TAMSG='DATA' FREAD FNAME)%RESET
<15>  $>(30$EQ$EX3$TAMSG)%NULL
<16>  MSG
<17>  $>0
<18> NULL:DATA=$.0
<19>  $>UNLOCK
<20> RESET:
<21> $*@@@ INCREMENT LINE POINTER @@@
<22>  MSG='RNUM' FGETLINE FNAME
<23>  MSG=(RNUM+1000) FSETLINE FNAME
<24> $*@@@ UNLOCK IMPLICIT MTS LOCKING @@@
<25>  RESULT=DATA
    "

    " RESULT=MTSSTAT A;MSG;FNAME;LINE;FNUM
<1>  $*@@@ FIND LAST LINE IN USE IN FILE @@@
<2>   FNAME=,FNAMES<(FNUMS$EQA)%$.$,FNUMS;>
<3>   MSG=FMTS 'GETLST ',FNAME INTO 'LINE'
<4>   $>(12$EQ$EX3$TAMSG)%EOF
<5>   RESULT=,LINE/1000
<6>   $>0
<7>  EOF:RESULT=,1
    "

    " MTSUNTIE A;Y;FNAME;MSG
<1>  $*@@@ UNTIE FILE GIVEN BY A @@@
<2>   $>((FNUMS$.A)$GT$,FNUMS)%ERROR
<3>   FNAME=,FNAMES<(FNUMS$EQA)%$.$,FNUMS;>
<4>  $*@@@ UPDATE FNUMS AND FNAMES @@@
<5>   FNUMS=(Y=FNUMS$EPA)%FNUMS
<6>   FNAMES=Y$C1FNAMES
<7>   $>0
<8>  ERROR:'ATTEMPTING TO UNTIE AN UNTIED FILE'
    "
```

```
    " A MTSWRITE B;FNUM;RNUM;MSG;FNAME;M;DATA
<1>   FNUM=1$TAB
<2>   FNAME=,FNAMES<(FNUMS$EQFNUM)%$.$,FNUMS;>
<3>  $*@@@ SET APL INTERNAL FORMAT @@@
<4>   MSG='INTERNAL' FSETTYPE FNAME
<5>  $*@@@ SET DIRECT ACCESS MODIFIER @@@
<6>   #IO=0
<7>   M=32$,0
<8>   M<30>=1
<9>   MSG=M FSETMODS FNAME
<10>  #IO=1
<11>  $>(($,RNUM=1$DRB)$EQ0)%SEQ
<12> $*@@@ FIND LINE NO. FOR DIRECT ACCESS WRITE @@@
<13>  MSG=(RNUM*1000) FSETLINE FNAME
<14> SEQ:$>(0$EQ$EX3$TAMSG=A FWRITE FNAME)%RESET
<15>  MSG
<16>  $>0
<17> RESET:
<18> $*@@@ INCREMENT LINE POINTER @@@
<19>  MSG='RNUM' FGETLINE FNAME
<20>  MSG=(RNUM+1000) FSETLINE FNAME
    "

    " A $"FI1 B;X
<1>   X=1$TAB
<2>  $*@@@ BRANCH TO FUNCTION GIVEN BY B @@@
<3>   $>((X$EQ1),(X$EQ2),(X$EQ4),(X$EQ9),(X$EQ10))%L1,L2,L4,L9,L1
<4>   'NOT VALID $"FI1 COMMAND'
<5>  L1:A MTSCREATE 1$DRB
<6>   $>0
<7>  L2:A MTSWRITE 1$DRB
<8>   $>0
<9>  L4:MTSERASE A
<10>  $>0
<11> L9:MTSFUNTIE A
<12>  $>0
<13> L10:A MTSFTIE 1$DRB
    "

    " Z=$"FI2 B;X
<1>   X=1$TAB
<2>  $*@@@ BRANCH TO FUNCTION GIVEN BY B @@@
<3>   $>((X$EQ3),(X$EQ7),(X$EQ8))%L3,L7,L8
<4>   'NOT VALID $"FI2 COMMAND'
<5>  L3:Z=MTSREAD 1$DRB
<6>   $>0
<7>  L7:Z=MTSFNAMES
<8>   $>0
<9>  L8:Z=MTSFNUMS
    "
```

```
    " $"FI3 B
<1>  $*@@@ CALL FUNCTION GIVEN BY B @@@
<2>   MTSDEL 1$DRB
    "

    " Z=A $"FI4 B
<1>  $*@@@ CALL FUNCTION GIVEN BY B @@@
<2>   Z=MTSSTAT A
    "

    " $"LIBDELETE FNAME;MSG;X
<1>  $*@@@ DELETE FILE NAME FROM LIBNAMES @@@
<2>   MSG='INTERNAL' FSETTYPE 'LIBNAMES'
<3>   $>(0$EQ$EX3$TAMSG='X' FREAD 'LIBNAMES')%OK
<4>   MSG
<5>   $>OUT
<6>  OK:X=( (FNAME,(17-$,FNAME)$,' ')&.$EQ$TRX)$C1X
<7>  $*@@@ RESET RECORD NUMBER TO ONE @@@
<8>   MSG=FRELEASE 'LIBNAMES'
<9>   MSG='INTERNAL' FSETTYPE 'LIBNAMES'
<10>  $>(0$EQ$EX3$TAMSG=X FWRITE 'LIBNAMES')%OUT
<11>  MSG
<12> OUT:MSG=FRELEASE 'LIBNAMES'
    "

    " $"LIBUPDATE FNAME;STATUS;MSG;X
<1>  $*@@@ CHECK IF LIBNAMES ALREADY CREATED @@@
<2>   $>(0$EQ$EX3$TAMSG=FMTS('CHKACC LIBNAMES') INTO 'STATUS')%RE
<3>  $*@@@ CREATE LIBNAMES AND PERMIT FOR READ @@@
<4>   MSG=FMTS 'CREATE LIBNAMES'
<5>   MSG=FMTS 'SHARE LIBNAMES'
<6>   X= 0 17 $,' '
<7>   $>WRITE
<8>  READ:MSG='INTERNAL' FSETTYPE 'LIBNAMES'
<9>   MSG='X' FREAD 'LIBNAMES'
<10>  MSG=FRELEASE 'LIBNAMES'
<11> $*@@@ UPDATE LIBNAMES @@@
<12> WRITE:X=X,<1> FNAME,(17-$,FNAME)$,' '
<13>  MSG='INTERNAL' FSETTYPE 'LIBNAMES'
<14>  MSG=X FWRITE 'LIBNAMES'
<15>  MSG=FRELEASE 'LIBNAMES'
    "
```

```
    " Z=$"LIB1 LIBNO;STATUS;MSG;FNAME
<1>  $*@@@ THIS FUNCTION RETURNS A LIST OF FILENAMES @@@
<2>  $*@@@ CONTAINED IN THE ACCOUNT GIVEN BY LIBNO   @@@
<3>   $>(0$EQ$,LIBNO)%NOACC
<4>   FNAME=(1$DR5$TALIBNO),':LIBNAMES'
<5>   $>NEXT
<6>  NOACC:FNAME='LIBNAMES'
<7>  NEXT:$>(0$EQ$EX3$TAMSG=FMTS('CHKACC ',FNAME) INTO 'STATUS')%
<8>   Z= 0 17 $,' '
<9>   $>0
<10> READ:MSG='INTERNAL' FSETTYPE FNAME
<11>  MSG='Z' FREAD FNAME
<12>  MSG=FRELEASE FNAME
    "

    " Z=$"MTSID LIST
<1>  $*@@@ THIS FUNCTION FINDS THE USER ID @@@
<2>  $*@@@ FOR A LIST OF USERS             @@@
<3>   Z=#AV<#IO+$TR(4$,256)$EN,LIST>
    "

    " R=$"REVMTSID USER
<1>  $*@@@ THIS FUNCTION FINDS THE USER ID @@@
<2>  $*@@@ IN ENCODED FORM FROM CHAR REPN  @@@
<3>   R=256$DE(#AV$.USER)-#IO
    "

    " $"TRAP1 LABEL
<1>  $*@@@ THIS FUNCTION DOES NOTHING. IT IS USED TO REMOVE @@@
<2>  $*@@@ THE ERROR TRAPPING FACILITIES PROVIDED BY #TRAP  @@@
    "

#  FNAMES
C $,$,$EQ 2; $,$EQ 0 16

#  FNUMS
N $,$,$EQ 1; $,$EQ 0
```

Transliteration Table

ALPHA	α	$AL	*LESS OR EQUAL*	≤	$LE
ALPHABET	*A-Z*	A-Z	*LESS THAN*	<	$LT
	a-z	a-z _A-_Z	*LINE FEED*		$¬
AND	∧	&	*LOCKED FUNCTION*	⍎	$L"
ASSIGN	←	=	*LOGARITHM*	⍟	$@ $LO $LN
BRACKETS	[	<	*MAXIMUM*	⌈	$MA $CE
	]	>	*MEMBERSHIP*	∈	$EP $ME
BRANCH	→	$> $GO	*MINIMUM*	⌊	$MI $FL
			MINUS	-	-
CAP	∩	$CA	*MODULUS*	|	$|
CIRCULAR FUNCTIONS	○	$$ $CI $PI	*MULTIPLY*	×	*
COLON	:	:			
COMMA(CATENATION)	,	,	*NAND*	⍲	$N&
COMMENT	⍝	$* $CO	*NEGATION*	-	$-
COMPRESSION	/	%	*NOR*	⍱	$N| $NR
COMPRESSION[1]	⌿	$C1	*NOT*	~	¬ $NO
CUP	∪	$CU	*NOT EQUAL*	≠	$NE
			NULL	⍬	$:
DECODE	⊥	$DE $BA			
DEFINITION(DEL)	∇	"	*OMEGA*	⍵	$OM
DELTA	Δ	$"	*OR*	∨	| $OR
DELTAU	⍙	$U"			
DIERESIS	¨	$DS	*PARENTHESES*	(	(
DIGITS	0-9	0-9		)	)
DIMENSION	⍴	$, $RH	*PERIOD*	.	.
DIVIDE	÷	/	*PLUS*	+	+
DROP	↓	$DR $DO			
			QUAD	⎕	#
ENCODE	⊤	$EN $RP	*QUAD - DIVIDE*	⌹	$/
EQUAL	=	$EQ	*QUOTE - QUAD*	⍞	$#
ESCAPE	⍂	$OUT	*QUOTE*	'	'
EXPANSION	\	$%			
EXPANSION(1)	⍀	$X1	*RANDOM*	?	?
EXPONENTIATION	*	@	*ROTATION*	⌽	$RO $RV
EXECUTE	⍎	$EX	*ROTATION*[1]	⊖	$R1
			RSUB	⊃	$RS
FACTORIAL	!	$FA			
FORMAT	⍕	$FM	*SEMICOLON*	;	;
			SUB	⊂	$SU
GRADE DOWN	⍒	$GD	*SYSTEM*	⌶	$IB $SY
GRADE UP	⍋	$GU			
GREATER OR EQUAL	≥	$GE	*TAKE*	↑	$TA $UP
GREATER THAN	>	$GT	*TRANSPOSITION*	⍉	$TR
INDEX	⍳	$. $IO $IN	*UNDERSCORE*	_	$UN

From UBC APL