Programming
For
Concurrency Control In Database Systems

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Master of Social Sciences in Computer Science

at the

University of Waikato

by

Tan, Puay Hiang

(*20th February, 1989*)

UNIVERSITY OF WAIKATO

Abstract

Concurrency control problems in centralized Database Management Systems have been actively studied in past two decades. Various types of concurrency control mechanisms have been proposed and implemented in practice. Among these mechanisms, Two-Phase Locking, Timestamping and Optimistic mechanisms have attracted most attention. This thesis presents a survey on these three mechanisms, identifies their major problems and ways to resolve these problems.

Despite their popularity, literatures published to date on these three mechanisms are mostly theoretical in nature, discussions on their implementation issues are normally neglected. This thesis attempts to look into this aspect by investigating the use of concurrent programming techniques based on semaphore and monitor in their implementations. Detailed descriptions of the implemenations are given and various modifications to the concurrent programming techniques to improve their applicability in the DBMS environment are provided too.

Acknowledgements

I wish to thank Professor E.V. Krishnamurthy for his valuable suggestions and encouragement during the course of this work. I am also indebted to Dr. Olivier de Vel for his supervision while Prof. Krishnamurthy was away. Above all, thanks to Siew Soh Mei for her constant encouragements and supports.

TABLE OF CONTENTS

Chapter 1
INTRODUCTION

1.1 Importance of DBMS

Over the last two decades, information has grown to be an essential resource in most organizations, such as financial institutions, military, airlines, etc. These organizations are very much depend on the *availability* and *accessibility* of the information for decision making in their daily organizational operations and management. Effective management of data resources will provide an organization a fast and easy access of the required information. This in turn will increase the efficiency and performance of an organization. With the advances in computing technology, these goals can be achieved by developing computer-based systems to manage the data resources. Such software system which manages a collection of data resources (or *database*) is called **Database Management System (DBMS)**.

Database management system allows an organization to have centralized control of its data resources. This feature provides the users a fast and easy access of the data resources as well as the following advantages [DATE86] :

- Inconsistency can be controlled.
- Data in the database can be shared by multiple users.
- Database integrity can be maintained.
- Redundancy in the database can be reduced.
- Standards in storing data can be enforced.
- Database security can be applied.

Due to these advantages and the rapid growth of data resources required to be managed, DBMS has become a very essential tool to most organizations. Its reliability and trustworthness practically affect the operations and management of

these organizations. For instance, any data corruption in a database could mean thousands of dollar loss to a bank account holder or destruction of human lives by incorrect military fire mission.

1.2 The need for Database Concurrency Control

Allowing multiple users to share data resources is one of the major advantages offered by the DBMS. However, this may also result in unauthorized access of crucial information (such as government confidential information) or violation of database integrity constraints and consistency when concurrent accesses are performed. Therefore, certain mechanisms are required in a DBMS to protect the database from unauthorized access and prevent it from being corrupted.

To prevent the violation of integrity constrants and preserve the consistency of a database, concurrent access to the database must be properly controlled. For example, without proper synchronization, two users may have reserved a same last seat on a flight by accessing the airline reservation database concurrently. Mechanisms to prevent the occurrence of such problem are so called **Database Concurrency Control Mechanisms**. Similar concurrency problems occur in the Operating Systems as well. In fact, the underlying concepts of most existing database concurrency control mechanisms are heavily based on the research done in the Operating Systems.

1.3 Objectives

Two-Phase Locking and Timestamping mechanisms are widely accepted in existing DBMSs. However, literatures published to date on these two mechanisms are mostly theoretical in nature, techniques in implementing them are usually neglected. This thesis attempts to look into the implementation aspects of these two mechanisms by using concurrent programming techniques based on *semaphore* and *monitor*. Beside these two mechanisms, Optimistic concurrency control mechanism which adopts an entirely different approach from the other two is briefly discussed.

The objectives of this thesis are summarised as below.

1. Studying three major concurrency control mechanisms, namely Two-Phase Locking mechanism, Timestamping mechanism and Optimistic mechanism, in a centralized DBMS environment;
2. Investigating the use of concurrent programming techniques based on *semaphore* and *monitor* in implementing Two-Phase Locking mechanism and Timestamping mechanism.
3. Suggesting modifications to the concurrent programming techniques to enhance their applicability in the domain of database concurrency control.

1.4 Organization

This thesis is organized into five major chapters.

Chapter 1 gives a brief introduction to Database Management Systems, presents the need for Database Concurrency Control in DBMS and outlines the objectives of this thesis.

Several basic concepts, such as database consistency and integrity constraints, are provided in chapter 2. This chapter also presents three database concurrency control problems and two major approaches to avoid these problems.

Chapter 3 surveys on three database concurrency control mechanisms, namely Two-Phase Locking mechanism, Timestamping mechanism and Optimistic mechanism. Correctness criteria of these three mechanisms and their strategies in resolving the concurrency control problems are discussed. Furthermore, detailed discussion on various synchronization problems, such as deadlock and livelock, are given in this chapter too.

Chapter 4 provides a detailed description of the use of concurrent programming techniques based on *semaphore* and *monitor* in implemeting Two-Phase Locking and Timestamping mechanisms. Several modifications on these techniques are suggested in this chapter to enhance their applicability in the domain of DBMS. This chapter also provides detailed algorithms for implementing the two mechanisms in near PASCAL format.

In conclusion, chapter 5 summarizes the works presented in chapter 2 to 4, discusses the concurrency control problems in distributed DBMS environment

and the future trend of applying other concurrent programming techniques in the implementation of database concurrency control mechanisms.

Chapter 2
Database and Concurrency

2.1 Transaction Processing

Transaction Processing (or sometimes referred to as *Transaction Management*) is a very important activity in Database Management System (DBMS).The application of transaction processing in DBMS provides a good environment in solving several database system problems, in particular the problems of **Concurrency Control**. Different schemes based on this approach have been developed to resolve the database concurrency control problems in the past two decades. In this section, the fundamental concept of the transaction processing, definition of transaction and their application in solving concurrency control problems in DBMS will be discussed and examples will be given as appropriate.

2.1.1 Database Integrity Constraints and Consistency

Before introducing the notion of transactions and transaction processing, definitions of the terms *database integrity constraints and consistency* are given.

a. What is database integrity constraint ?

Database integrity constraints are the assertions or rules that the **entities**[1] (or objects) in the database must not violate. This in turn will ensure correctness and accuracy in reflecting the real world. For example, an integrity constraint maintained in a next-of-kin database is *"The father must be older than his children"*. Any update or insertion of information that violate this constraint will cause the incorrectness in the database and consequently create confusions in the

[1] *Entity :- Any distinguishable object that is to be represented in the database [DATE86].*

subsequent operations. Therefore, ensuring that the entities do not violate the integrity constraints is one of the major tasks in database systems.

b. What is database consistency ?

A database is said to be in *consistent state*, if all entities in that database satisfy the integrity constraints. For example, consider the transfer of money from one account to another in the banking system. If a user of the system transfers \$100 from his saving account to his cheque account, there is an instance between the two actions while the \$100 is debited from the saving account but not yet credited into his cheque account. The database state at this particular instance is *inconsistent* as it violates the integrity constraint, namely *'the total amount of money in the system should remain constant as long as there is no money withdrawn from/deposited to the system'*. Such situation may be depicted schematically by *figure 2.1.*

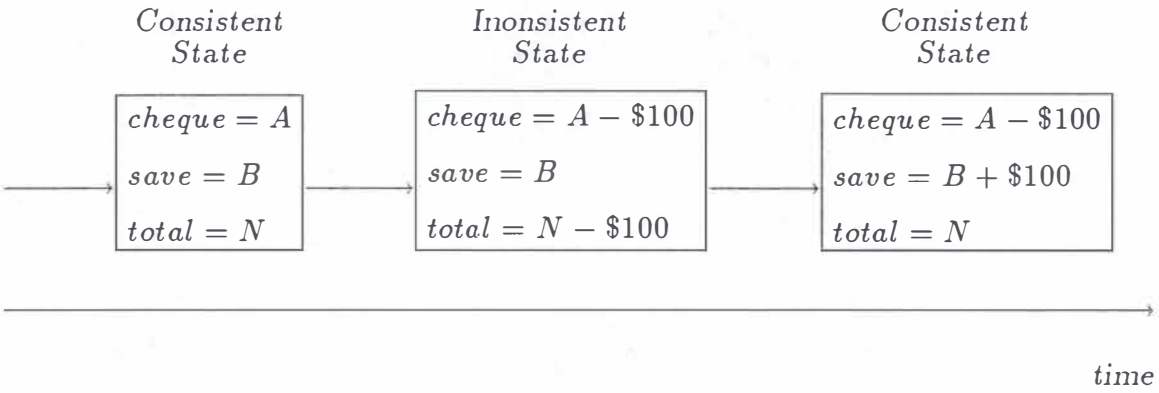


Figure 2.1 Database state transition

In practice, this type of temporary inconsistent state transition is always unavoidable. A program that implements such an operation is considered as **correct** as long as it brings the database from a consistent state to another consistent state[2] after the operation. (*See figure 2.2*)

[2] *The new database state can be identical to the state before the operation or entirely different.*

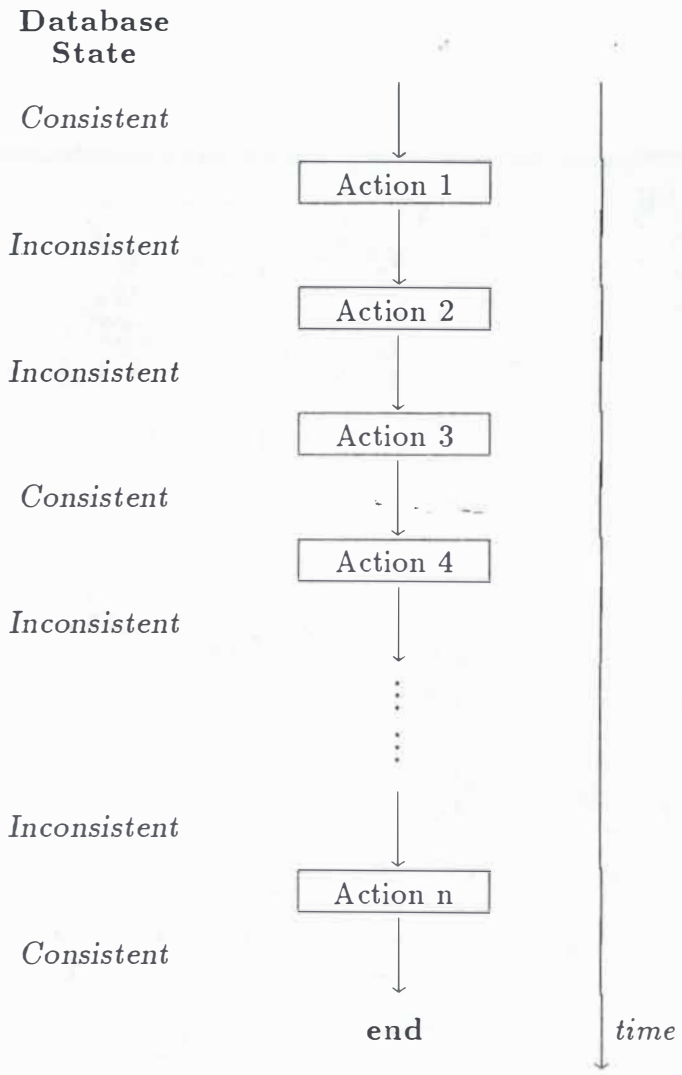


Figure 2.2 A correct operation consists of inconsistent state transitions

Based on this definition, execution of a correct operation in isolation guarantees database consistency at the end of the operation. However, such feature may not hold when several correct operations are executing simultaneously, as in a *multi-users shared database system*. If a DBMS allows several users to execute their operations and simultaneously access a same piece of data, the database may get into an inconsistent state at the completion of the operations; example below illustrates this problem.

Example. *Suppose we have two entities* **A** *and* **B** *in the database and the only integrity constraint is that* $A = B$. *Consider there are two users* **U1** *and* **U2** *executing the following operations.*

$$\mathbf{U1}: A := A - 10 \; ; \; B := B - 10 \; ;$$

$$\mathbf{U2}: A := A \times 10 \; ; \; B := B \times 10 \; ;$$

Both operations will bring the database from consistent state to consistent state if they are executed one after the other. However, if the two operations are executed at the same time and interleaved in the following sequence,

U1: $A := A - 10$; **U2:**$A := A \times 10$; **U2:**$B := B \times 10$; **U1:**$B := B - 10$;

the database will definitely get into inconsistent state with $\mathbf{A} \neq \mathbf{B}$ *at the completion of both operations. For instance, suppose the initial values of both* **A** *and* **B** *are 1 which satisfy the integrity constraint* **A=B**. *After the executions of both operations according to the above sequence,* **A** *and* **B** *become -90 and 0 respectively which obviously violates the integrity constraint and putting the database into inconsistent state.*

2.1.2 Transaction

A **transaction** is a logical unit of work which consists of a sequence of atomic actions (eg. *read, write, add, etc.*). Each transaction when executed alone, transforms a consistent state into another consistent state without necessary preserving consistency at all intermediate stages [ESWA76] [DATE83]. In the other words, *transactions are correct operations which guarantee the preservation of database consistency.* (*See figure 2.3*).

In a DBMS that supports transaction processing, a transaction has yet another important property :

> *The execution of a transaction is either terminated successfully with all its actions performed on the database committed*[3] *or aborted without modifying the database state.*

[3] *Once an action on a database is committed, it is guaranteed never to be undone. [DATE83]*

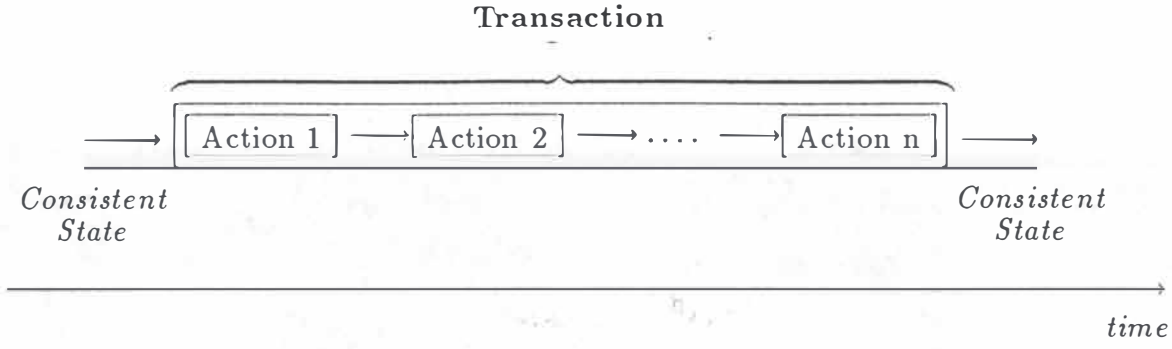


Figure 2.3 A transaction

With this property and appropriate synchronization, the DBMS can ensure the **preservation of consistency** while several transactions are executed simultaneously.

A transaction may be in the form of an on-line query expressed either in a database query language or an application program written in programming language.

2.1.3 Transaction Processing

Since the successful implementation of the *American Airline SABRE* reservation system in early 1960's, **transaction processing** has become a very important concept in both Operating Systems and DBMS. It introduced a whole new series of considerations into both areas and opened a new branch of *Real-Time Database Systems*[4].

Transaction processing can be further divided into two tasks, namely *scheduling transactions* and *managing system recovery*.

a. Scheduling concurrent transactions

In multi-user database systems based on transaction processing, there is a component called **concurrency controller** (or **scheduler**) which responsible for

[4] *Real-Time Database System are characterised by their immediate response to the external users. For example, request for money withdrawal in a banking system must be responded in seconds to prevent the customers from unnecessary waiting.*

scheduling the concurrent transactions. The concurrency controller accepts transactions from different users and schedules their atomic actions (or data base requests) in a sequence that will preserve the database consistency at the completion of the transactions. The output sequence will be then sent to the other module of the database system - *data manager* - which manages the actual database. *Figure 2.4* shows the atomic actions of a set of n transactions being scheduled into a serial schedule.

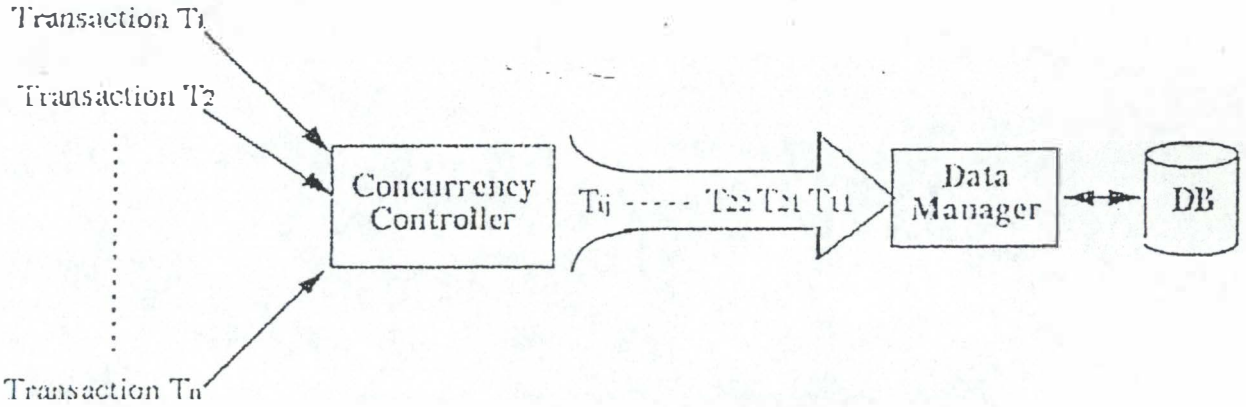


Figure 2.4 Concurrency Controller

(where T_{ij} = jth atomic action of the ith transaction)

An inappropriate scheduling methodology will always create *concurrency control problems* and lead the database into an inconsistent state. In the past decade, different scheduling methodologies (or *concurrency control mechanisms*) have been studied and developed to support the concurrency controller. Study of the underlying principles of these methodologies and investigation of the use of *concurrent programming techniques* for their implementation are the two primary objectives of this thesis. Further details of different types of scheduling methodologies[5] will be given in later chapters.

b. Managing system recovery

System failures due to external factors, such as power failure, can happen any time while large amount of transactions are being processed in the system. Restoring

[5] *In the rest of the thesis, the terms scheduling methodology and concurrency control mechanism will be used interchangeable.*

the database back to a consistent state and starting the transactions that have been aborted during the failure are two major problems the component responsible for system recovery has to deal with. System recovery is a very different theoretical and pratical problem and so will not be studied in this thesis. Reference is made to [GRAY78,HAE83,LORIE77,GRAY81] for more information on this topic.

2.2 Concurrency Control Problems

2.2.1 Introduction

> **"Concurrency - the existence of several simultaneous or parallel activities."**[LISTER]

Concurrency is a naturally inherent feature of the world. At all time, each individual human being is performing some tasks while the others are performing some other tasks. Some of these tasks are *interdependent* which have to be carried out one after the other and some are *independent* that can be done concurrently without worrying about what others are doing or have done. Synchronizing the concurrent tasks correctly by using computer has been a popular problem in the computing community for a long time. Intensive studies and research have been done and number of techniques have been proposed and implemented to resolve various problems arise while concurrency is involved.

Introducing concurrency into DBMS is essential, especially in this information intensive era. It definitely improves the system's response time, information utilization and transaction throughput. However, it introduces problems in synchronizing concurrent transactions as well. Inappropriate synchronization may lead to the loss of updates performed by the transactions, provision of incorrect information to the users and violation of database integrity constraints. These problems are so called **Concurrency Control Problems.** In order to ensure the correct execution of the concurrent transactions, a DBMS requires some mechanisms to control the occurrence of these problems. In this section, three concurrency control problems , which commonly occur in existing DBMSs with inappropriate (or without) concurrency control mechanism, will be discussed with examples provided.

2.2.2 Problem 1 - Loss of Updates_

Suppose a customer, say *A*, tries to withdraw \$100 from his saving account via automated teller machine (*ATM*). If at the same time, at another *ATM*, a joint account holder, say *A*'s wife (*AW*), also tries to withdraw \$200 from the same account. Without an appropriate concurrency control, one of these withdrawals may not be reflected in the final account's balance. The two transactions may be scheduled in the sequence shown in *figure 2.5* and result in an incorrect update on the account.

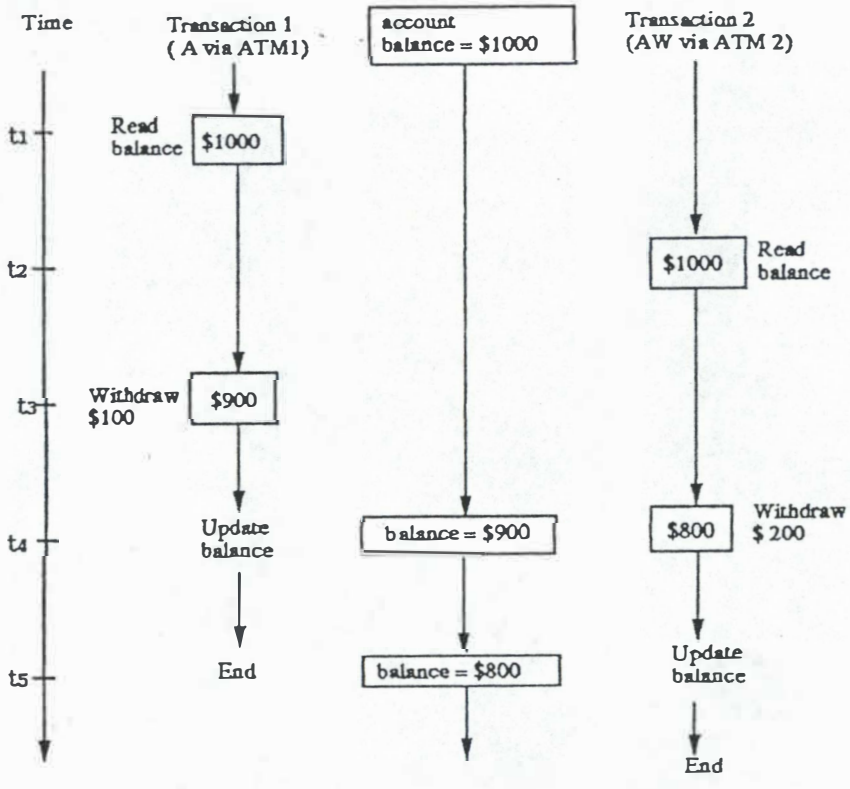


Figure 2.5 Concurrent withdrawal from same account

Consider there is \$1000 in the saving account initially. *ATM1* reads the account balance which is \$1000 from the database at time *t1*; at time *t2*, *ATM2* reads the account balance which is still \$1000 from the database; *ATM1* deducts \$100 from the \$1000 it read and gives \$100 cash to *A* at time *t3*; at time *t4*, *ATM2* deducts \$200 from \$1000 it read and gives *AW* \$200 cash; at the mean time, *ATM1* updates the new balance it calculated which is \$900 to the database; however this newly updated balance is overwritten by the update of \$800 account balance done

by *ATM2* at time *t5*. At the end of both transactions, the final balance of the saving account is \$800 instead of \$700. The incorrect account balance is due to the update of transaction 2 at time *t5*. In other words, the update done by transaction 1 at time *t4* is lost. This problem is called **Lost Update Problem or write-write conflict**.

2.2.3 Problem 2 : Reading of Phantom Object

Consider the example of the automatic banking system again. Suppose *A* tries to withdraw \$100 from his saving account via *ATM1*, but being rejected due to the lack of available cash at *ATM1*. Also assume that at the same time, his wife *AW* tries to withdraw \$200 from the same account via *ATM2*. One of the following two undesirable consequences may arise if the database system does not equip with a concurrency control mechanism.

Consequence 1 : *AW may not be able to withdraw her* \$200 *even though there is sufficient fund in her account.*

Suppose there is \$200 *in the saving account and the two transactions initiated by A and AW are scheduled in the sequences shown in figure 2.6.*

Fortunately, the depicted undesirable consequence will not bring the account into inconsistent state. This is because the roll back operation after the abortion of the transaction 1 will bring the account balance back to the balance before transaction 1 is executed. In the other words, AW will be able to withdraw her \$200 *from the account if she tries again. However, the delay caused to AW and other people who are waiting for AW to finish her transaction can be avoided if a concurrency control mechanism is available.*

Consequence 2 : *Incorrect balance is updated.*

Suppose there is \$1000 *in the saving account and the transaction initiated by A and AW are scheduled in the sequence depicted in figure 2.7.*

As shown in figure 2.7, instead of \$800, \$700 *is being updated as the final account balance even though only* \$200 *has been withdrawn. Obviously, this undesirable consequence is more severe than the previous one.*

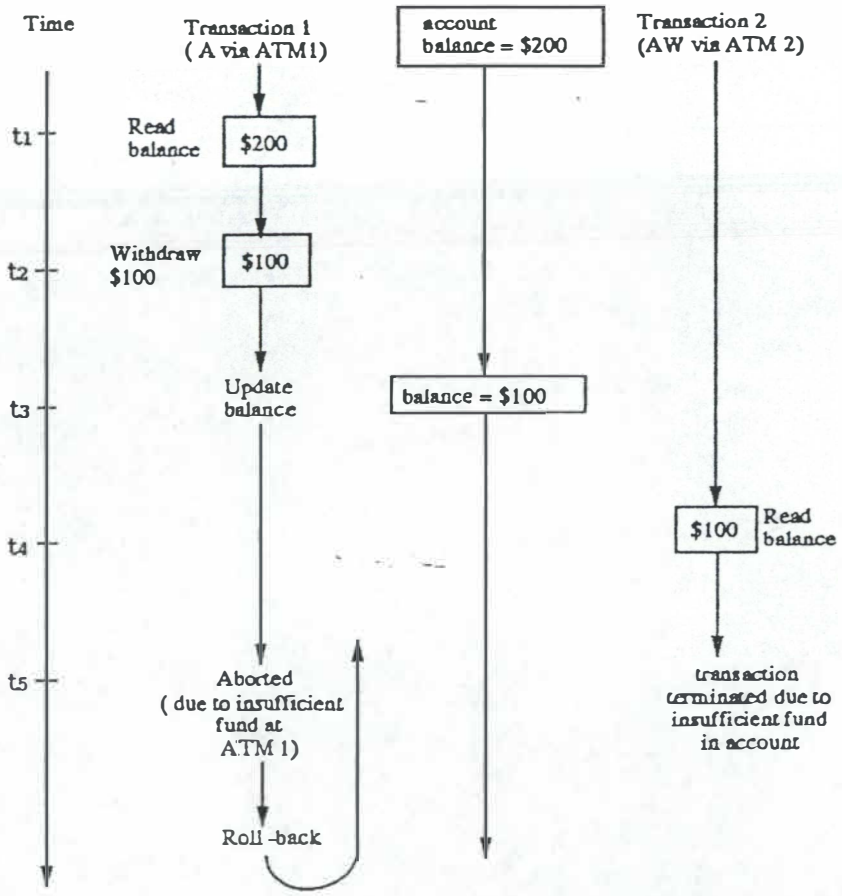


Figure 2.6 Reading of Phantom Object - consequence 1

Both undesirable situations described above are caused by reading the uncommited update made by transaction 1. The uncommited updated data object which later being undone is called **phantom object** - object that does not actually exist. Reading of phantom object is sometimes referred to as **dirty reading or wite-read conflict.**

2.2.4 Problem 3 : Inconsistent Retrieval

This problem is probably not as serious as the last two problems, but is still considered as undesirable and unnecessary. Suppose *A* is transferring $500 from his saving account to his cheque account via *ATM1* and his wife *AW* is checking the total balance of their accounts via *ATM2* simultaneously. *AW* may obtain an incorrect total balance if the two transactions are being scheduled as shown in *figure 2.8*.

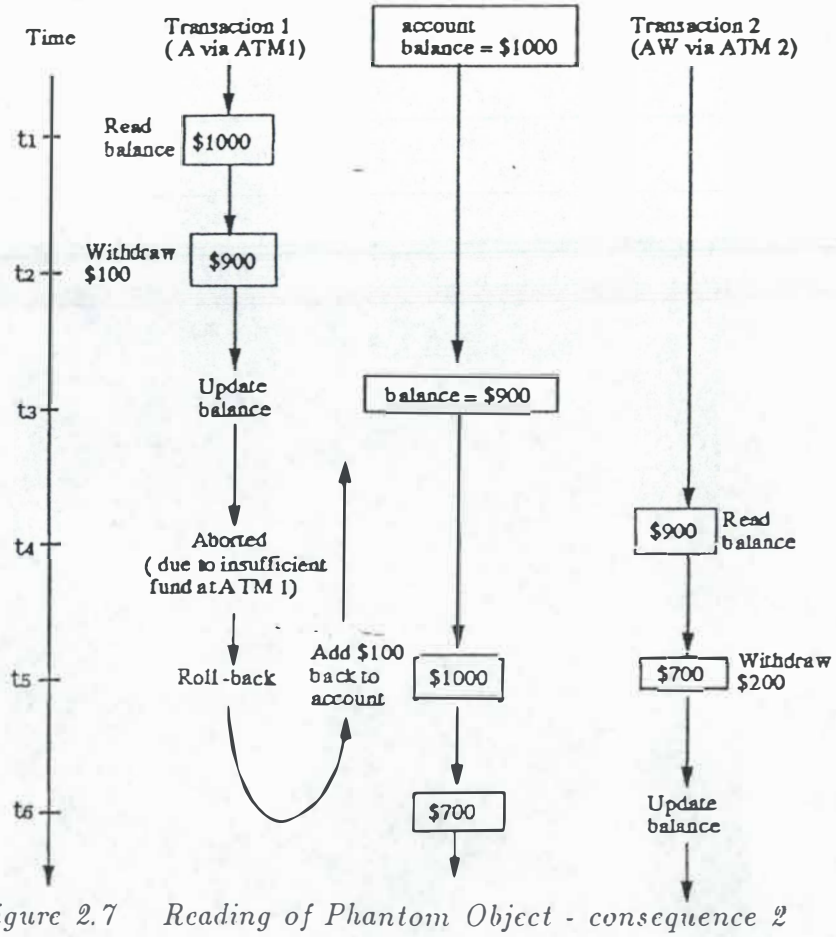


Figure 2.7 Reading of Phantom Object - consequence 2

At the completion of both transactions, transactions 1 is successfully terminated leaving the database in a consistent state; but transaction 2 is terminated with incorrect total balance returned to *AW*. The problem with the schedule is that transaction 1 is allowed to update the accounts' balances and change the database state while transaction 2 is reading the accounts' balances. As shown in *figure 2.8*, transaction 2 read the saving account's balance at time *t2* and read the cheque account's balance after transaction 1 has terminated with all updates committed at time *t8*. In other words, transaction 2 has accessed the database in two different database states. Even though both database states are consistent, they are inconsistent with respect to each other. Therefore, accessing data in two different database states leads to the inconsistent retrieval. Such problem is called **inconsistent retrieval** or **read-write conflict**.

2.2.5 Remarks

The three concurrency control problems described in the last few subsections can be expressed as a *transaction conflict matrix* as shown in *figure 2.9*. We assume

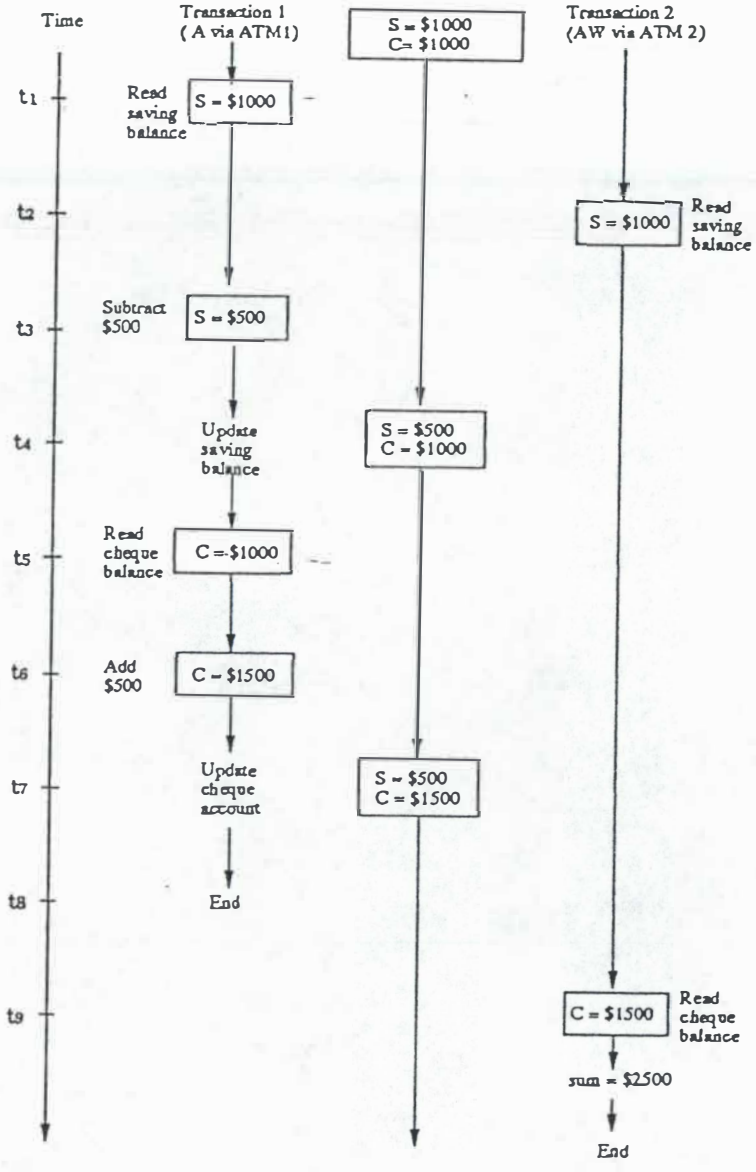


Figure 2.8 Inconsistent Retrieval

that transaction 1 has its database action executed on an object before transaction 2 does on the same object.

As illustrated in the conflict matrix, there are four possible combinations of database actions on a same object from two transactions, which are *read-read*, *read-write*, *write-read* and *write-write*. Beside the combination of *read-read*, the rest correspond to the three situations where three concurrency control problems could occur. There is no conflict between two transactions with read accesses on a same object, since read access will never change the database state.

Note that the occurrence of the three concurrency control problems are either

Transaction 1 \ *Transaction 2*	Read	Write
Read	No Conflict	Conflict *(problem 3)*
Write	Conflict *(problem 2)*	Conflict *(problem 1)*

Figure 2.9 Transaction Conflict Matrix

due to accessing database while the database state is changing or putting the database into inconsistent state at the completion of the transactions. Based on these observations, the following conclusion is derived.

> "*No conflict will occur between two simultaneously executing transactions as long as there is no write access on an object from one transaction and a read/write access on the same object from another transaction.*"

In the other words,

> "*A conflict exists between two simultaneously executing transactions if both transactions access a same object and at least one of these accesses is a write access.*"

2.3 Correctness

A *transaction* is considered as correct since it preserves the database consistency at the completion of its execution. However, such correctness cannot be guaranteed when more than one transactions are concurrently executing in a shared database system. Different types of problem as the three problems mentioned earlier may arise and put the database into an inconsistent state or give the users an inconsistent view. Therefore, the major task of a concurrency control mechanism is to schedule the atomic actions of the transactions submitted to the database system in a sequence that it will preserve the database consistency and provide the users a consistence view.

Basically, two approaches are available to ensure the correctness of a schedule. These are (i) *syntactic approach* and (ii) *semantic approach.*

2.3.1 Syntactic Approach - Serialization Approach

Syntactic approach requires only the syntactic information [6] about the transactions to ensure the correctness of the schedule. By using the syntactic information, a correct schedule can be produced such that the final effect of its execution on the database is equivalent to executing a sequence of transactions one after the other. Executing a set of transactions one after another always preserves the database consistency at the end of the execution, since each transaction executing alone always preserves database consistency (see *figure 2.10*). Due to this property, such approach is also called **Serialization Approach.**

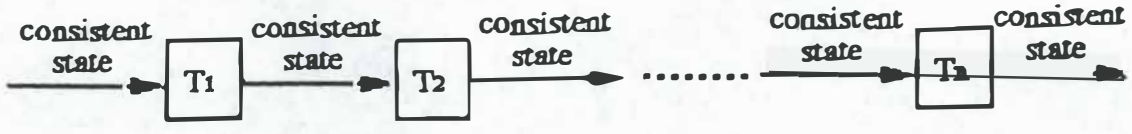


Figure 2.10 Serial Execution of Transactions

[6] *Such as the order of the atomic actions in each transaction, names of the objects accessed by the transactions and types of access (read or write).*

Based on this approach, the schedules produced will preserve the database consistency but not necessary provide a consistent view (*see example given in section 2.2.4*). Due to this reason, the correctness criteria of the serialization approach is in fact based on a more restricted class of definition - **conflict serialization.**

A schedule is considered as conflict serializable [7], *if*

i. it is equivalent to a serial schedule;

and

ii. all pairs of conflicting database accesses are scheduled in the same order as in the corresponding serial schedule;

(*where two database accesses are conflicting if they are accessing a same object and at least one of them is a write access* [8].)

The following example shows how a conflict can arise in a serializable schedule.

Example. *Consider the two transactions given in section 2.2.4.*

$T1$	=	$Read(S)$	$T2$ =	$Read(S)$
		$S := S - 500$		$Read(C)$
		$Write(S)$		$Sum := S + C$
		$Read(C)$		End
		$C := C + 500$		
		$Write(C)$		
		End		

where S and C are the saving and cheque accounts' balances respectively;

The schedule given in the same section is,

$S1$	=	$T1$:	$Read(S)$
		$T2$:	$Read(S)$
		$T1$:	$S := S - 500$
		$T1$:	$Write(S)$

[7] *In the rest of the thesis, all serializable schedules are considered as conflict serializable.*

[8] *A similar definition as conflict transactions given in subsection 2.2.5, but at the level of transaction step.*

$T1$: $Read(C)$

$T1$: $C := C + 500$

$T1$: $Write(C)$

$T2$: $Read(C)$

$T2$: $Sum := S + C$

which is equivalent to a serial schedule $S = T2T1$[9] *as both* S *and* $S1$ *terminated with a same consistent database state. Nevertheless,* $S1$ *is not conflict serializable since the conflicting database accesses* $T1$: $Write(C)$ *and* $T2$:$Read(C)$ *in* $S1$ *are not scheduled in the same order as in* S. *By reversing the order of* $T1$: $Write(C)$ *and* $T2$:$Read(C)$ *in the schedule* $S1$, *we will get a new schedule* $S2$ *which is conflict serializable.*

$S2$ = $T1$: $Read(S)$

$T2$: $Read(S)$

$T1$: $S := S - 500$

$T1$: $Write(S)$

$T1$: $Read(C)$

$T1$: $C := C + 500$

$T2$: $Read(C)$

$T1$: $Write(C)$

$T2$: $Sum := S + C$

We may apply this new schedule to the example provided in section 2.2.4 and find that it preserves the database consistency and returns a correct total balance - \$2000 - *to AW.*

One major advantage of the serialization approach is its *generality*. This approach can be applied into any application domain without any modification because it does not require any semantic information about the transactions involved nor the integrity constraints imposed on that domain. In the other words, it is *application independent*. However, as pointed out by Kung and Papadimitriou

[9] *In fact,* $S1$ *is equivalent to serial schedule* $T1T2$ *too, but only* $T2T1$ *is considered here to avoid the confusion when conflict serializability is introduced later in the example.*

in [KUNG79], the correctness criteria of serialization approach is weaker than that of semantic approach, which means that the set of correct schedules(SR) produced by the concurrency controllers based on serialization approach is smaller than the set of correct schedules (SM) produced by the concurrency controllers based on semantic approach, ie. $SR \subset SM$.

Two-Phase Locking, Timestamping and Optimistic approaches are all considered as serialization approach.

2.3.2 Semantic Approach - Nonserialization Approach

As indicated by Kung and Papadimitriou[KUNG79],

> "*The more information available to the schedulers, the 'better' scheduling results may be expected.*"

This statement is especially true when semantic information about the transactions and the integrity constraints are available prior to the scheduling. Kung and Papadimitriou also showed that an approach using semantic information and integrity constraints has a stronger correctness criterion than that of the serialization approach. This is because the former recognizes serializable schedules as well as schedules that are correct but not serializable. Due to this feature, semantic approach is also called **Non-serialization Approach.**

Now, the problem is how does the semantic information can be utilized to ensure the correctness of a schedule. If the semantic information about the transactions and integrity constraints is available , this problem is rather similar to *parallel program verification*, since the schedule has to be verified to preserve an *invariant*, namely the *database consistency.*

Several approaches [ASH75,KELLER76,WOLF85] have been suggested to verify the correctness and properties of parallel programs by using assertions. These approaches are mainly based on the idea proposed by Robert W. Floyd [FLOYD67] :

> *".... an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition will be true at that time. Then by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left by a connection whose associated proposition will be true at that time. By this means, we may prove certain properties of programs, particularly properties of the form :*
>
> > *"If the initial values of the program variables satisfy the relation R1, the final values on completion will satisfy the relation R2."*
>
> *....."*

In the context of DBMS, a schedule can be thought as a parallel program, each transaction participated in that schedule can be treated as a serial program in the parallel program, and the propositions associated with the connections are the integrity constraints. Therefore, the following property can also be proved,

> "*If the initial database state is consistent, the database state at the completion of the schedule will be consistent.*"

which is in fact the preservation of database consistency. Based on this same idea, Lamport proposed the following scheduling policy in [LAMP76] :

> "*... the request to execute one step in a transaction is granted only if the execution will not invalidate any of the assertions attached to those arcs where the tokens of other transactions reside at that time.*"

Lamport represented a transaction as a flowchart of atomic actions and the execution of the transaction as a token flowing through the flowchart. By following the above scheduling policy, the arrival of a token at the output arc of the flowchart denoted a successful termination of the corresponding transaction without violating any integrity constraint. If all participating transactions are executed in this way, the database consistency will be preserved and the correctness of the schedule is ensured.

Another interesting approach to verify the properties of a parallel program was suggested by Wolfgang[WOLF85]. He showed that the properties of a parallel program represented by a *Predicate/Transition net* (P/T net) can be verified by the *S*-invariant of that net. If one can model a set of application programs accessing the database as a P/T net, the integrity constraints imposed on the database can also be verified by using the *S*-invariant of that net. As long as all the integrity constraints are verified, database is guaranteed to be in a consistent state at the termination of each transaction.

Generally, a concurrency controller based on semantic approach consists of a set of application programs where an instantiation of each program's execution corresponds to a transaction. It synchronizes the concurrent transactions based on the knowledge of the integrity constraints and semantic information about the transactions and thus produce a correct schedule. A transaction is normally activated by the arrival of the users' requests. For example, based on the scheduling policies proposed by Lamport and Wolfgang, a transaction is activated by the arrival of a "*request token*" from the users. A database access is permitted only when at least one (or more depends on the weight of the arc in the case of Wolfgang's approach) token is available on each input arc associated with this access.

Despite its stronger correctness criteria, semantic approach is not commonly used to implement the concurrency controller in DBMS due to the following disadvantages :

1. Substantial amount of time will be needed to prove the assertions; this lengthens the transaction's execution time which in turn reduces the response time and the transaction throughput.
2. It is *application dependent.* Hence, concurrency controller based on the semantic approach can only be used in the application area where the semantic information about the transaction as well as integrity constraints imposed on that application domain are known to the controller. In the other words, concurrency controller has to be reorganized to remove the old set of semantic information and introduce a new set of information each time when it is applied to a different application domain.
3. Usually, semantic information about the transactions as well as integrity constraints are not always available to the designer of the concurrency controller.

Therefore, syntactic approach is widely preferred in existing commercially available DBMSs.

Chapter 3
Mechanisms for Concurrency Control

3.1 Introduction

In the last two decades, number of concurrency control mechanisms have been proposed. Some of these have been implemented in existing commercially available DBMSs. The majority of these mechanisms use syntactic approach rather than semantic approach due to the undesirable features of the latter as earlier mentioned (*see section 2.3.2*). In this chapter, three mechanisms which are commonly adopted to implement the concurrency controllers are introduced and studied in detail. The three mechanisms are :

- **Two Phase Locking**
- **Timestamping**
- **Optimistic mechanism**

Generally, a concurrency control mechanism based on syntactic approach has the basic structure shown in *figure 3.1*. Upon receiving a request to access an object in the database from a user, the controller will schedule this request by checking it against other previous requests on the same object. The request will be then output to the data manager if it passes the test, otherwise it will join the delaying queue. Each successful request may invalidate certain conditions that cause the previous denial of some requests on the delaying queue, therefore the requests on the delaying queue have to be checked again after each successful request.

A factor that differ one concurrency control mechanism from another is the checking criteria imposed in '*check*' procedure. The size of the class of correct schedules that a concurrency control mechanism can produce depends very much on the strictness of these criteria. On the other hand, the amount of syntactic

```
procedure concurrency-controller ;
begin
  loop
    receive(request-to-access-object) ;
    check(request-to-access-object,ok) ;
    if ok then
    begin
      send-to-data-manager(request-to-access-object) ;
      request := head-of-delaying-queue ;
      while delaying-queue-is-not-empty do
        begin
          check(request,ok) ;
          if ok then
          begin
            send-to-data-manager(request) ;
            remove-request-from-queue(request) ;
          end ;
          request := next-request-on-delaying-queue ;
        end ;
      end
    else join-delaying-queue(request-to-access-object) ;
  forever
end ;
```

Figure 3.1 General structure of concurrency controller

information about a transaction that a concurrency controller can obtain at one time and when the information are acquired play a very important role in the concurrency controller too. There are two types of *information acquisition* that are commonly used in existing DBMSs, namely *Dynamic acquisition* and *Pre-transaction acquisition.*

• Dynamic acquisition

Concurrency controller obtains the syntactic information about a transaction's request upon its arrival. As a transaction proceeds without any *hindrance* (*eg. deadlock in Two-Phase Locking mechanism*), the concurrency controller can gain more syntactic information about this active transaction until the last request of the transaction is received.

- **Pre-transaction acquisition**

All required syntactic information about a transaction are available on the arrival of the first request of that transaction; ie. once a transaction is activated, a list of data objects accessed by this transaction and their types of access (*read or write*) have to be sent to the concurrency controller.

Both types of acquisition have their advantages and disadvantages. Pre-transaction acquisition introduces an additional overhead. This arises due to the fact that each activated transaction has to prepare a list of data objects to be accessed before sending its requests to the concurrency controller. However, this strategy also permit the concurrency controller to foresee and avoid the problems that may arise in the transaction. (*More will be discussed on this argument with examples later*). Concurrency controller adopting dynamic acquisition will not be able to foresee and avoid the problems that may arise prior to the execution of the transaction. This is due to the insufficient syntactic information available (*See definition of Dynamic Acquisition on page 26*). Problems are normally not detected till they have occurred. However, dynamic acquisition allows the concurrency controller to recognize and produce a bigger class of correct schedules than that output by the controllers using pre-transaction acquisition. Examples to illustrate this feature will be given later in this chapter.

3.2 Two Phase Locking

This technique of locking is widely used in Operating Systems to avoid multiple processes from accessing a single global resource. By using *lock* and *unlock* operations, a process is guaranteed to have an exclusive possession of a global resource while it is operating on it. Similar concept has been adopted in DBMS to avoid undesirable problems while more than one transactions are accessing a same data object (*see section 2.2*). Of course, the technique of locking cannot be directly applied to the concurrency controller without any modification because a DBMS and an Operating System are two entirely different environments.

A concurrency controller that adopts *locking* technique locks a specified data object and blocks other requests from accessing this object until the object is unlocked. When a request to access a data object is being blocked in the concurrency controller, the execution of the transaction from which the request is originated will too be blocked until the request is granted. In short, the tasks of the concurrency controller are to insert *LOCK* and *UNLOCK* in the input schedule and to ensure that no request on a locked object is granted.

Basically, a locking mechanism has the following constraints :-

a. a transaction's request submitted-to the concurrency controller must hold a *lock* on the specified data object before it can be output to the data manager;
b. all data objects accessed by a transaction must be locked before any unlocking

.

Due to the second constraint, a *locking* mechanism is also called **Two- Phase Locking**. Execution of a transaction can be partitioned into two phases where the first phase is the *locking phase* (or *growing phase*) and the second phase is the *unlocking phase* (or *shrinking phase*). (See *figure 3.2*).

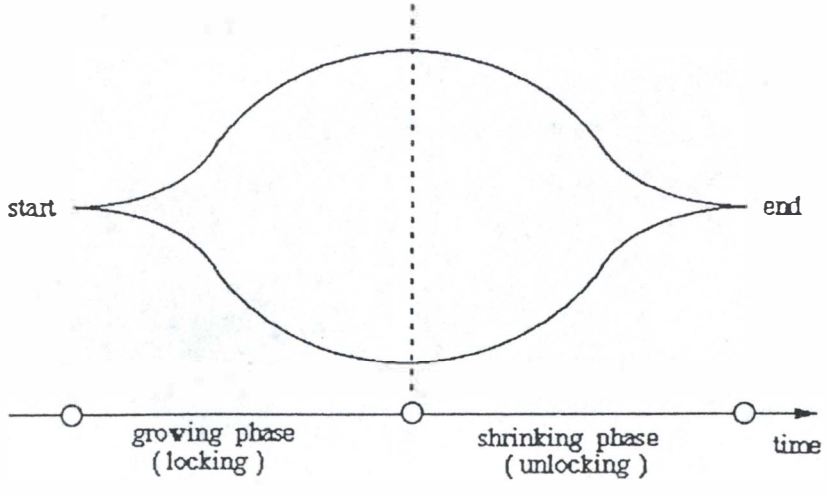


Figure 3.2 Growing and Shrinking of a transaction

Unfortunately, the above Two-Phase Locking constraints are insufficient to support an efficient concurrency controller. As explained in *section 2.2.5*, read

accesses on a same data object from two different transactions do not create any conflict and therefore can proceed independently. In other words, a transaction's request to read a data object which is locked by another read access shall not be delayed but granted immediately. Obviously, *lock* alone is not sufficient to support such a feature. To solve this problem, *lock* operation is classified into two different types of *lock*, namely **read lock** and **write lock**.

3.2.1 Read Lock and Write Lock

Read lock and write lock are sometimes referred to as *shared lock* and *exclusive lock* respectively. With these two types of lock, the locking protocol can be revised as follows :-

- A *read* request must obtain a *read lock* on the specified data object before it is granted;
- A *write* request must obtain a *write lock* on the specified data object before it is granted;
- If a data object is locked with a *read lock* by transaction *T1*, then a different transaction *T2* can acquire a *read lock* on the same data object but not otherwise;
- If a data object is locked with a *write lock* by transaction *T*, no other transactions can acquire a *read* or *write lock* on the same object till the object is unlocked by *T*.

One type of *lock L1* (*read or write*) is considered as *compatible* with another type of *lock L2* if both are *read locks*; otherwise they are *incompatible*. The compatibility of the *read* and *write locks* can be expressed as a *compatibility matrix* shown in *figure 3.3*.

By considering the transaction *T1* as holding transaction and *T2* as requesting transaction, *compatibility matrix* is isomorphic to the transaction conflict matrix shown in *figure 2.9*. Therefore, we can rephrase the third and fourth rules of the locking protocol as follows :-

Transaction T2

Transaction T1		Read lock	Write lock
	Read lock	*Compatible*	*Incompatible*
	Write lock	*Inompatible*	*Incompatible*

Figure 3.3 Compatibility Matrix

- *Transaction T2 can acquire a lock on a data object which is currently locked by transaction T1 if the acquisition of the lock does not create any conflict between T1 and T2.*

Based on this rephrased rule, Two-Phase Locking mechanism avoids conflicts by checking the compatibility of the requesting lock and the holding lock. It also avoids the three concurrency control problems correspond to the three conflicts in the conflict matrix shown in *figure 2.9*. Nevertheless, this does not mean that Two-Phase Locking is problem-free. It suffers from a serious problem, namely the **DEADLOCK**.

3.2.2 Deadlock

Supposing that a transaction *T1* is holding locks (*read or write*) on some data objects and requesting to lock a data object O_i which is currently locked with an incompatible lock by another transaction *T2*. Execution of *T1* will be blocked as its request to lock O_i is delayed in the concurrency controller until *T2* unlock O_i. Unfortunate situation may arise when *T2* is attempting to lock a data object O_j which is one of the data objects currently locked with incompatible lock by *T1*. Therefore, both requests from *T1* and *T2* are being delayed *indefinitely* in the concurrency controller and execution of both *T1* and *T2* are blocked indefinitely as well. Such situation is called **deadlock**. This problem gets more complicated when the number of transactions involved in such a '*circular waiting ring*' is large.

In fact, deadlock is a very common problem in our daily life. The traffic jam is one such example of the deadlock problem. A commonly seen traffic jam during the peak hour of a working day is depicted in *figure 3.4*. Supposing that there are four streams of traffic flowing from four directions as shown in the figure. Each car uses a portion of the road exclusively (*ie. lock*). There are four intersections (*A, B, C and D*), each is a junction of two of the traffic streams. Deadlock will occur when each traffic stream is indefinitely blocking an intersection. (See *figure 3.4*)

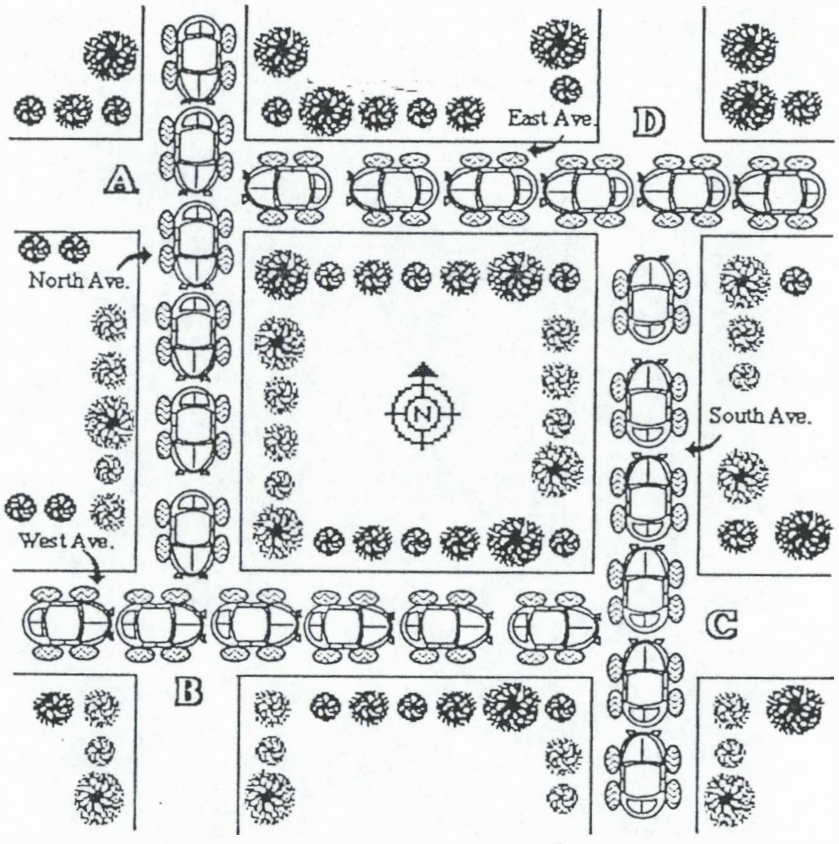


Figure 3.4 Traffic jam problem

A traffic stream (say the Northward bound traffic) can be thought as a transaction which is exclusively holding a lock on a data object (*intersection C*) while waiting to lock another data object (*intersection D*).

The problem of deadlock is extensively studied in Operating Systems and various strategies are used to resolve this problem. Two of these are commonly used to solve the deadlock that occur in concurrency control mechanism. They are *(i) preclaim strategy* and *(ii) pre-emptive strategy.*

(i) **Preclaim Strategy**

This is a *deadlock prevention strategy*. This strategy requires the concurrency controller to obtain locks on all data objects accessed by a transaction prior to the execution of the transaction. No transaction request is granted until all locks on the required data objects have been acquired. Obviously, this strategy is best for *pre-transaction Two-Phase Locking* mechanism. Prevention of deadlock by using this strategy will be demonstrated with an example later while pre-transaction Two-Phase Locking is described.

(ii) **Pre-emptive Strategy**

Here, an algorithm is required to detect the occurrence of deadlock. Once a deadlock is detected, one of the *deadlocked* transactions has to be *pre-empted* by releasing all the locks that it is holding and terminating the transaction. All updates that have been done on the database by the pre-empted transaction have to be undone to restore the database to a consistent state. As transaction recovery is not in the scope of this thesis, we refer to [GORD86] for more information on this topic.

Several methods are available to detect deadlock. The best known among these is probably the construction of **conflict graph** (*or wait-for-graph*). Conflict graph is a directed graph used to indicate the conflicts among the active transactions. Each node in the graph represents an active transaction and each directed arc represents a conflict between two transactions. A node is added to the graph when the first request of a transaction is received ; and removed with its associated arcs from the graph when the last request of the corresponding transaction is granted. A directed arc is added from one node T_i to another node T_j when T_j is requesting a lock on a data object which is locked by T_i with an incompatible lock. In other words, an arc indicates a conflict between two transactions and the transaction represented by the *destination* node must *wait* for the transaction represented by the *source* node to unlock the data object involved. A deadlock will occur only when a newly added arc results in a *directed cycle* in the conflict graph.

Such a cycle indicates a waiting ring where each transaction in the cycle is indefinitely waiting for another transaction to release the lock on the requested data object. Therefore, the deadlock can be detected by checking the conflict graph for directed cycle after each new arc is added. *Figure 3.5* shows a simple conflict graph with deadlock. The directed cycle $T1 \rightarrow T2 \rightarrow T4 \rightarrow T1$ indicates a deadlock where $T1$ is waiting for $T4$, $T4$ is waiting for $T2$ and $T2$ is waiting for $T1$.

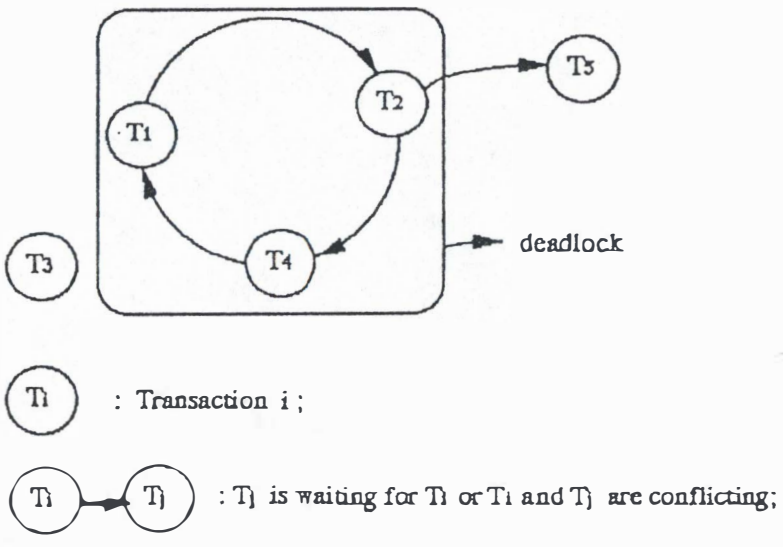


Figure 3.5 Conflict Graph with Deadlock

A simple way to resolve a deadlock is to *abort* one of the *deadlocked* transactions and remove its corresponding node from the conflict graph. Once such a node is removed, at least two of the arcs that form part of the cycle will also be removed leaving the graph *cycle-free* (*and also deadlock free*) again.

3.2.3 Livelock

Besides deadlock, Two-Phase Locking suffers from another problem - **Livelock.** *Figure 3.6* shows a simple example of livelock.

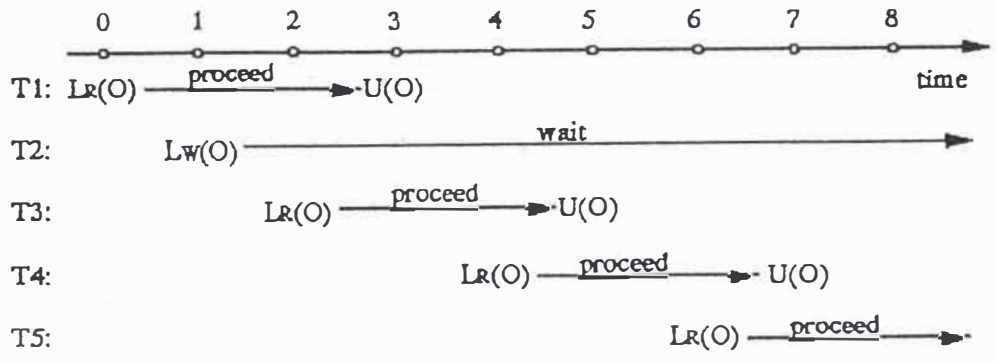


Figure 3.6 -- Livelock

Suppose that a data object O is read-locked by transaction $T1$ at time 0 and transaction $T2$ is requesting a write-lock on the same object O at time 1. Since the two locks are conflicting, $T2$'s request to lock O ($Lw(O)$) is delayed and the transaction is blocked until O is unlocked. Unfortunately, while $T2$ is waiting for object O to be unlocked, $T3$'s request to read-lock O is granted at time 2 since its lock is not conflicting with the lock $T1$ is holding on O. In other words, $T2$ has to wait for both $T1$ and $T3$ to unlock O before its request to write-lock O can be granted. $T2$ may get into a situation where more and more requests to lock an object are granted while it is waiting for that object to be unlocked. Such situation is called livelock.

Livelock can be avoided by queuing the delayed transaction requests and attempting (*granting or delaying*) each newly arrived request with the following additional rule :-

A request to lock a data object shall not be granted, if

1. *there is at least one request on the delayed queue associates with this object,*

and

2. *no request from the same transaction on this data object has been granted prior to this request;*

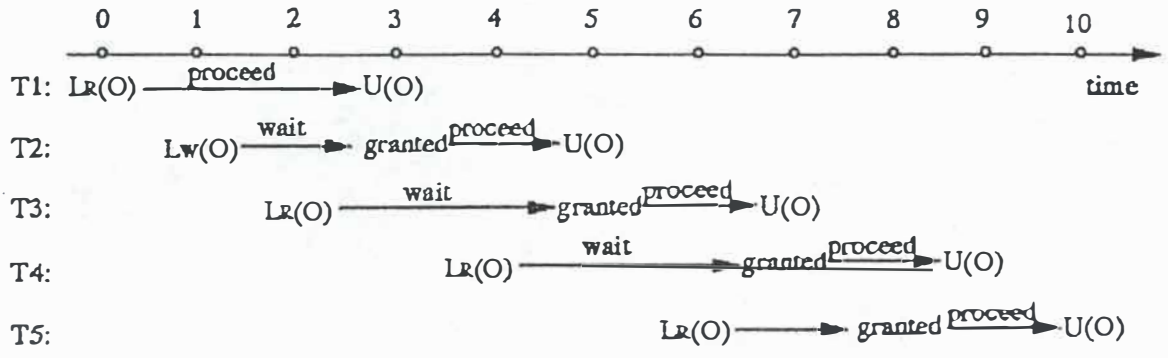


Figure 3.7 - Schedule without livelock

The livelock problem depicted in *figure 3.6* can then be solved by applying the above additional rule. *Figure 3.7* depicts a new schedule without livelock.

3.2.4 Pre-transaction Two-Phase Locking

Two-Phase Locking mechanism adopting *preclaim strategy* is considered as a *pre-transaction Two-Phase Locking mechanism*. The locking protocol of this mechanism is :

> *"Lock all the data objects accessed by a transaction prior to the arrival of the first request of this transaction and unlock all the data objects after the last request is granted."*

The following example shows how a Two-Phase Locking mechanism based on the above locking protocol can prevent the occurrence of deadlock.

Example. *Consider that the following schedule is input to a concurrency controller using pre-transaction Two-Phase Locking mechanism.*

T1 :	$R(x)$			$W(y)$		
T2 :		$R(y)$			$W(z)$	
T3 :			$R(z)$			$W(x)$

The above is a classic example of the schedule that will result in a deadlock situation. However, with pre-transaction Two-Phase Locking protocol, dead-

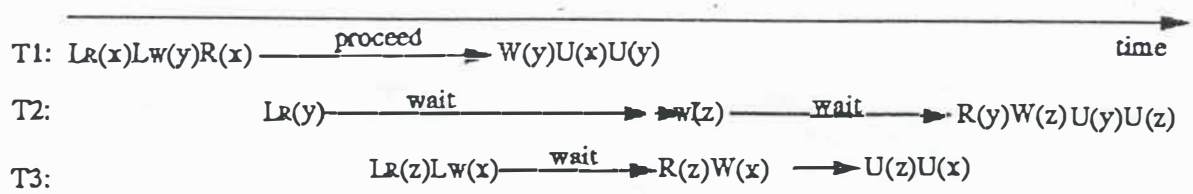


Figure 3.8 Deadlock-free schedule

lock situation will never arise. Figure 3.8 shows how the concurrency controller reschedule the above schedule into a deadlock-free correct schedule.

3.2.5 Dynamic Two-Phase Locking

Dynamic Two-Phase Locking mechanism is characterized by its dynamic acquisition of locks. Lock is acquired on a data object on each arrival of the transaction request. Its locking protocol is :

> "*Lock the data objects immediately before each request is made and unlock all the locked data objects after the last request is granted.*"

This is different from pre-transaction Two-phase-locking in which all the data objects must be locked prior to the submission of the first request. One major advantage of this mechanism is its ability to recognize and produce a larger class of correct schedules in comparison with the pre-transaction Two-Phase Locking. On the other hand, this mechanism faces the threat of deadlock. To resolve the problem of deadlock without losing its feature of dynamic locking, we need to equip it with a *deadlock detection mechanism* that dynamically checks the occurrence of deadlock.

In the following two examples, we demonstrate the advantage and disadvantage of this mechanism. The first example shows how a deadlock may occur and the second example presents a correct schedule which can be recognized by dynamic Two-Phase Locking mechanism but not pre-transaction Two-Phase Locking mechanism.

Example. *Supposing that the same input schedule presented in the last example (page 35) is submitted to a dynamic Two-Phase Locking concurrency controller. Figure 3.9 depicts how this schedule is reshuffled and how deadlock may occur.*

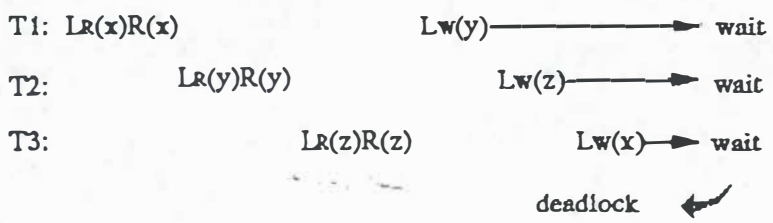
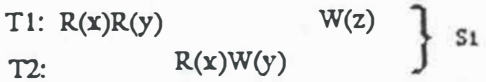


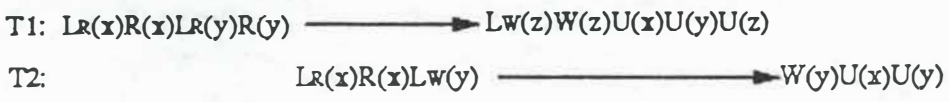
Figure 3.9 Deadlock in Dynamic Two-Phase Locking Mechanism

Deadlock occurred in the above schedule as each transaction is attempting to acquire write-lock on a data object which is currently read-locked by another transaction.

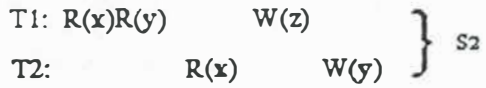
Example. *Consider the following schedule as an input schedule submitted to the concurrency controller.*



The output schedule produced is



which is



With the same input schedule (S_1), pre-transaction Two-Phase Locking mechanism reshuffle the schedule into the following schedule.

T1: R(x)R(y)W(z)
T2: R(x)W(y)
} S3

In other words, pre-transaction Two-Phase Locking mechanism can only recognize[10] *schedules S_1, S_2 and S_3 as S_3; but dynamic Two-Phase Locking mechanism can recognize the schedules S_1 and S_2 as S_2, S_3 as S_3.*

3.2.6 Granularity Issues

Granularity of a lock refers to the size of the lockable data object. The granularity of lock can be as large as a whole database or as small as a field in a record. Large and small granularities of lock are commonly referred as *coarse* and *fine* granularities respectively. Obviously, the *finer* the granularity of lock, the higher the degree of concurrency. On the other hand, the *coarser* the granularity of lock, the lower the chances that certain problems such as concurrency control problems, deadlock and livelock may arise. *Figure 3.10* shows the inter-relationship of the degree of concurrency, probability of problem occurrence and the granularity.

In the last few subsections, single granularity of lock is assumed, but this may not be the case in practice. Multiple granularities of lock are normally adopted in concurrency controllers. The reason for this is easy to understand. For transactions, such as enquiring the total balance of all saving accounts in a banking database, are more efficient with a coarser granularity which allows a bigger group of objects to be locked with a single lock operation. For other transactions, such as enquiring the balance of a single account, a finer granularity is better since it will reduce the number of unnecessary locks and thus increase the degree of concurrency. However, in practice, both types of transaction are always needed

[10] *Schedule S_i is recognized as schedule S_j, if S_i is submitted to the concurrency controller and S_j is the output schedule.*

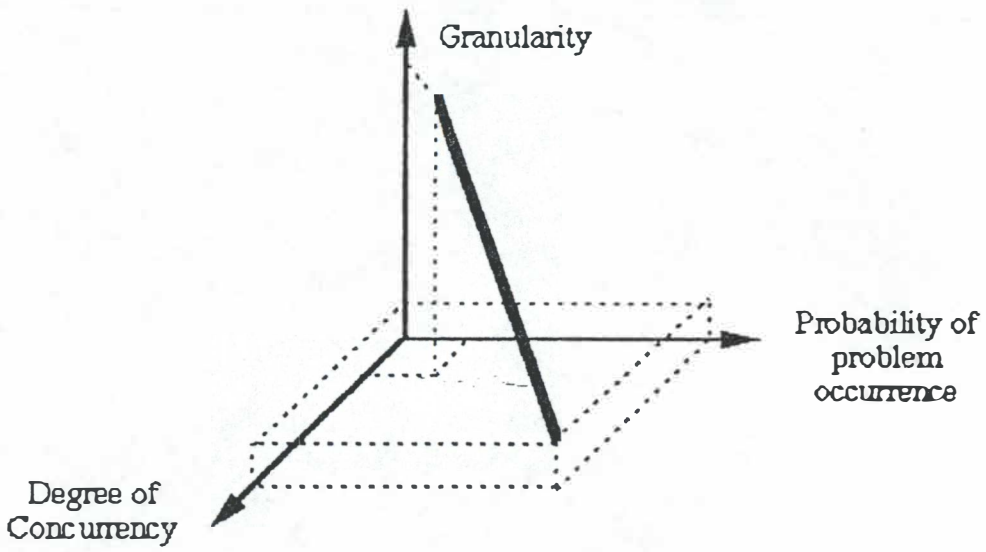


Figure 3.10 Inter-relationship of three factors

in a same DBMS. Therefore, a **multiply-granular lock scheme** is required to improve the performance of a concurrency controller.

To support a *multiply-granular lock scheme*, Gray [GRAY77] suggested a *hierarchical lock strategy*. In his strategy, the whole database is organized in a hierarchical structure, such as a tree. Each level of the hierarchy corresponds to a granularity of lock. For example, in *figure 3.11*, there are four different levels, each corresponds to *database, area, file* and *record*. Locks are permissible at each level. Therefore, a transaction to enquire the total balance of all saving accounts stored in a file can simply place a read lock on that file and disallow other requests to update any of the records in the file. This is much more efficient than locking each individual record.

Beside read lock and write lock, Gray introduced another two new locks[11],

[11] *Another new lock, namely 'share and intention write lock' (SIW) is not mentioned here since it is rather redundant. The purpose of having this lock is to allow the transaction which wants to read an entire subtree and to update particular nodes of that subtree to realize higher degree of concurrency[GRAY78]. However, locking the root of the subtree with a SIW lock is in fact equivalent to issueing two intention*

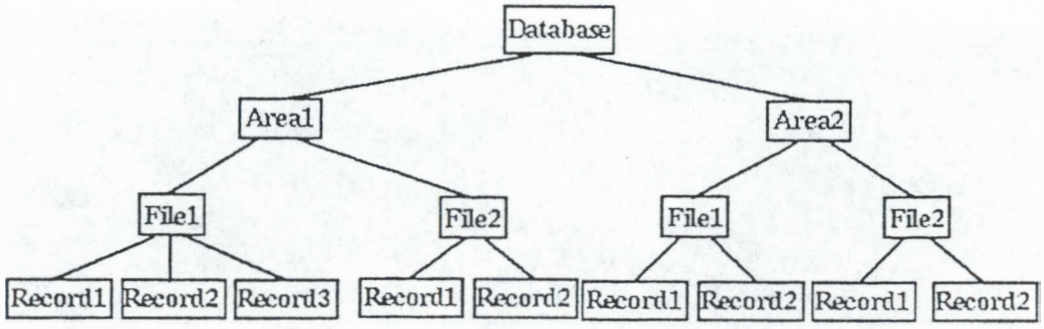


Figure 3.11 Tree structure of a database

namely intention read (IR) and intention write (IW) locks. When a transaction requests to read lock (or write lock) a data object in a database, it has to show its "*intention*" by locking all nodes along the path from the root node (the whole database in *figure 3.11*) to the immediate parent node of the destination node with *intention read lock* (or *intention write lock*). *Figure 3.12* depicts an example of locking a single record with read lock in the database shown in *figure 3.11*.

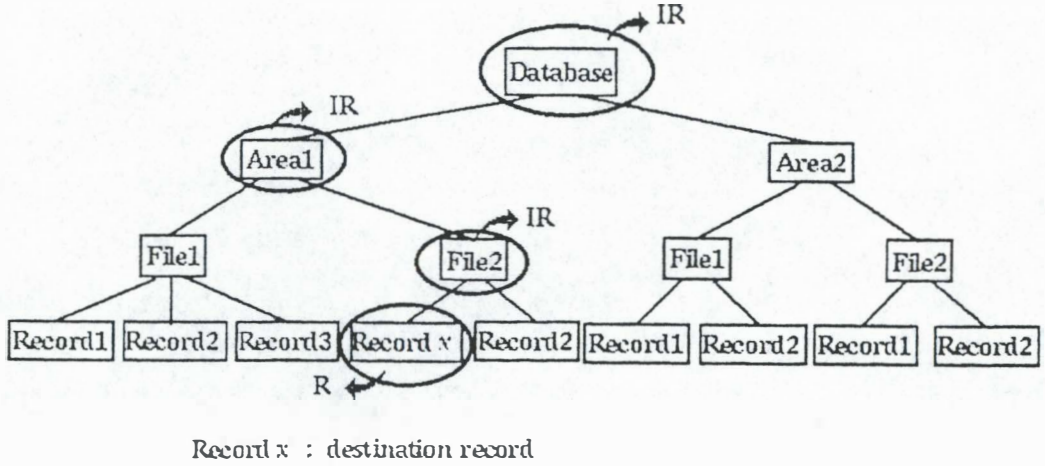


Figure 3.12 Example of locking a record with hierarchical locking

locks, IR lock and IW lock, from a same transaction. The descendent nodes that later identified as the nodes to be read (or updated) can then be locked with R (or W) locks.

Transaction T2 (requesting)

Transaction T1 (holding)	R	W	IR	IW
R	*yes*	*no*	*yes*	*no*
W	*no*	*no*	*no*	*no*
IR	*yes*	*no*	*yes*	*yes*
IW	*no*	*no*	*yes*	*yes*

where *yes* – Compatible

no – Incompatible

Figure 3.13 Revised Compatibility Matrix

With the two new types of lock introduced, the compatibility matrix of locks shown in *figure 3.3* has to be revised as *figure 3.13*.

At the end of each transaction, all locks (*intention or non-intention*) held by this transaction will be released from the finer granules up to the root.

3.3 Timestamping

In last subsection, concurrency control mechanism based on locking technique is discussed. Locking technique resolves the concurrency control problems but introduces another serious problem - *deadlock*. Detecting and resolving deadlock is a very expensive task but also unavoidable. Due to this reason, various *non-locking* techniques were proposed to develop *deadlock-free* concurrency controller. One such technique is the **Timestamping**.

Based on Timestamping technique, each transaction submitted to the concurrency controller is given a unique identifier called **timestamp**. Generally, timestamp of a transaction is the time instance at which the first request of that transaction arrives at the concurrency controller[12]. Hence, timestamp not only uniquely identifies a transaction, but also its '*age*'. Transaction *T1* is said to be '*older*' than transaction *T2* if *T1*'s timestamp is smaller than *T2*'s.

[12] *A timestamp is not necessarily the actual time of the system clock. An incremental count after each timestamp is issued is adequate to provide each transaction an unique identifier.*

By using the transaction's timestamp and other syntactic information (such as data objects being accessed and access types), a concurrency controller can schedule an input schedule into a *conflict serializable schedule.*

Several variations of Timestamping mechansim have been suggested. Two of these, *pre-transaction Timestamping* and *dynamic Timestamping* mechanisms, are discussed below.

3.3.1 Pre-transaction Timestamping

Pre-transaction Timestamping mechanism requires a transaction to provide syntactic information about this transaction prior to the first request being submitted to the concurrency controller. These information include the timestamp of the transaction, data objects accessed and the access types (*read or write*). With these information, the concurrency controller will be able to perform the following scheduling protocol on each subsequent request submitted.

> "*A request to access an object is granted if all conflicting requests from the older transactions been granted.*"

Take an example to illustrate how the above protocol is performed.

Example. *Consider that the follwoing schedule is input to the concurrency controller.*

T1 :	$R(x)$			$R(y)$				$W(x)$	
T2 :		$R(x)$							$W(y)$
T3 :					$W(z)$	$W(x)$			

Based on the pre-transaction Timestamping protocol, the output schedule is as follows :

T1 :	$R(x)$	$R(y)$			$W(x)$				
T2 :							$R(x)$		$W(y)$
T3 :				$W(z)$				$W(x)$	

1. *$T1 : R(x)$ is granted immediately since $T1$ is the oldest transaction.*
2. *$T2 : R(x)$ is delayed since the conflicting request $T1 : W(x)$ is not yet granted and $T1$ is older than $T2$.*
3. *$T1 : R(y)$ is granted immediately since $T1$ is the oldest transaction.*
4. *$T3 : W(z)$ is granted immediately since there is no conflicting requests from the older transactions ($T1$ and $T2$).*
5. *$T3 : W(x)$ is delayed since the conflicting requests $T1 : W(x)$ and $T2 : R(x)$ have not been granted and both $T1$ and $T2$ are older than $T3$.*
6. *$T1 : W(x)$ is granted immediately since $T1$ is the oldest transaction.*
7. *$T2 : R(x)$ delayed at stage (2) is now granted since all conflicting requests from the older transaction ($T1$) have been granted.*
8. *$T3 : W(x)$ delayed at stage (5) is now granted since all conflicting requests from the older transactions ($T1$ and $T2$) have been granted.*
9. *$T2 : W(y)$ is granted immediately since there is no conflicting request from the older transactions ($T1$ and $T2$) which has not been granted.*

From the above example, the following features are observed :

1. The older a transaction is, the faster its requests are granted.
2. Requests originated from a transaction will not be granted till all the conflicting requests from the older transactions are granted.

With these features, the following correctness criterion is always guaranteed by the pre-transaction protocol :

> *Schedules output from the concurrency controller are conflict serializable to the serial schedule*
>
> $$T_1 T_2 T_3 .... T_n$$
>
> *where*
>
> *T_i's are the transactions involved in the input schedule with timestamp i, and*
>
> *T_i is older than T_j, if $i < j$.*

In comparison with the correctness criterion ensured by the pre-transaction Two-Phase Locking, the above criterion is stronger; ie. the set of correct schedules recognized by a pre-transaction Timestamping concurrency controller is bigger than that recognized by a pre-transaction Two-Phase Locking concurrency controller[PAPA86]. The following example shows a correct input schedule recognized by the pre-transaction Timestamping mechanism but not by the pre-transaction Two-Phase Locking mechanism.

Example. *Consider the following conflict serializable schedule (s_1) is input to a concurrency controller.*

T1 :	$R(x)$	$R(y)$			$W(z)$	
T2 :				$R(x)$		$W(y)$

Based on the pre-transaction Timestamping mechanism, the same schedule is output. s_1 is being recognized as a correct schedule and output to the data manager. However, this is not so when s_1 is sent to a concurrency controller based on pre-transaction Two-Phase Locking mechanism. The output schedule will be :

T1 :	$R(x)$	$R(y)$	$W(z)$			
T2 :					$R(x)$	$W(y)$

s_1 is being recognized as a serial schedule T1T2 even though it itself is already a correct schedule.

3.3.2 Dynamic Timestamping

Based on *Dynamic Timestamping mechanism*, a conflict between two requests from two different transactions is not known to the concurrency controller till the younger conflicting request is accessing the data object. This type of dynamic acquisition of syntactic information may give rise to the deadlock problem in *Dynamic Two-Phase Locking mechanism*, but this problem will never occur in dynamic Timestamping mechanism. (*More about the problem of deadlock will be discussed later.*)

Beside the transaction, timestamps are also assigned to the data objects maintained in the database. Each data object in the database is given two different types of timestamp, namely *read-timestamp* and *write-timestamp*. *Read-timestamp* is the timestamp of the youngest transaction that last read the data object and *write-timestamp* is the timestamp of the youngest transaction that last updated the data object.

The scheduling protocol of this mechanism is as follows :

1. A request to read a data object O is granted if the timestamp of the transaction from which the request is originated is *larger than or equal to* the *write-timestamp* of O; otherwise, the request is delayed.
2. A request to write a data object O is granted if the timestamp of the transaction from which the request is originated is *larger than or equal to* both the *read-timestamp* and *write-timestamp* of O; otherwise, the request is delayed.

Notice that the mechanism based on the above protocol grants and delays requests in such a way that the conflicting requests from the older transaction are granted before the conflicting requests from the younger transaction. This ensures that the output schedules are *conflict serializable* to the serial schedule of all the transactions that are involved executed in their chronological order. This strategy also avoid the undesirable problem - *deadlock*.

3.3.2.1 Avoidance of Deadlock

In *section 3.2.2*, conflict graph is used to detect the deadlock in dynamic Two-Phase Locking mechanism. Same method can be applied here to find out if deadlock is possible in dynamic Timestamping mechanism. As shown previously (*see section 3.2.2*), if there is a cycle in the conflict graph, there is a deadlock in the schedule. Hence, if a cycle is impossible to be formed in a conflict graph, deadlock will never occur.

The following proof shows that deadlock is impossible in a dynamic Timestamping mechanism.

Proof :

The proof consists of two parts :

1. *Proving that a transaction T_i is older than transaction T_j if there is a path from the transaction node T_i to the transaction node T_j in the conflict graph.*
2. *Proving that a cycle is impossible to be formed without violating the first proof.*

Part 1 :

Based on the dynamic Timestamping protocol, if a conflict is allowed (both conflicting requests are granted), the older conflicting request must be granted before the younger conflicting request. In the other words, if there is a conflict arc from a transaction node T1 to a transaction node T2, T1 must be older than T2. Such that,

if $T1 \dashrightarrow T2$,

then $t1 < t2$

where ti is the timestamp of transaction Ti.

If there is another conflict arc from transaction node T2 to a transaction node T3, then T2 must be older than T3 based on the same reason. Such that,

if $T1 \dashrightarrow T2 \longrightarrow T3$,

then $t1 < t2, t2 < t3$

Hence, T1 must be older than T3 too where $t1 < t3$.

By induction, if there is a path from a transaction node Ti to another transaction node Tj, then Ti must be older than Tj. Such that,

if $Ti \Longrightarrow Tj$

then $ti < tj$.

where $\Longrightarrow$ represents a path in the conflict graph.

Part 2 :

Next, we have to prove that a cycle is impossible in a conflict graph constructed by the dynamic Timestamping mechanism. This is done by first assuming that there is a cycle in the graph and then prove its falsity.

By definition, a cycle in a directed graph is a path whose end-nodes coincide[CARR79]. Assume that there is such cycle in the conflict graph which comprise of a path from a transaction node T_i back to itself, such that $T_i \Longrightarrow T_i$. We

have already proved in part one that transaction T_j must be older than T_k if there is a path from transaction node T_j to transaction node T_k. Hence, if a path from node T_i back to itself does exist, then

1. *T_i is older than T_i.*
2. *All transactions correspond to the nodes involved in the cycle are older than T_i.*
3. *All transactions correspond to the nodes involved in the cycle are younger than T_i.*

which are all **IMPOSSIBLE**.

Therefore, we conclude that a cycle is impossible in the conflict graph constructed by the dynamic Timestamping mechanism. So, the mechanism is deadlock-free.

3.3.2.2 Livelock

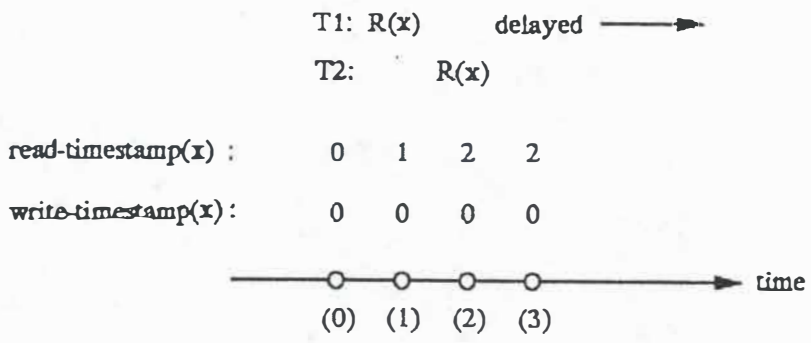
One major disadvantage of the dynamic Timestamping mechanism is that once a request is delayed, it will be delayed forever - which results in *livelock*. Based on the protocol presented earlier, a request is delayed if its transaction's timestamp is smaller than the read-timestamp (*request to read*) or both read-timestamp and write-timestamp (*request to write*). Since the timestamp is an ever increasing value, a younger transaction will never get a timestamp smaller than any older transaction. Hence, once a data object is accessed by a transaction, its read-timestamp (*read access*) or write-timestamp (*write access*) will never be smaller than the timestamp of this transaction. In the other words, delayed requests originated from the older transactions will not be able to proceed any further, since the access-timestamp (*read-timestamp and write-timestamp*) of the object will never get smaller than their timestamps. The following example shows how such problem may occur.

Example. *Consider that the following simple schedule is input to a concurrency controller using the dynamic Timestamping mechanism. (Assume that the read-timestamp and write-timestamp of all data objects are* 0 *initially.)*

$$\mathbf{T1}: \quad R(x) \qquad\qquad W(x)$$
$$\mathbf{T2}: \qquad\quad R(x)$$

T1 is older than T2 since the first request of T1 arrives at the concurrency controller before the first request of T2 does. Assume that T1 is given a timestamp 1 *and T2 is given a timestamp* 2.

The following figure illustrates the occurence of a livelock schematically.



(1) $T1 : R(x)$ *is granted immediately, since* $T1$*'s timestamp is larger than the write-timestamp of* x.

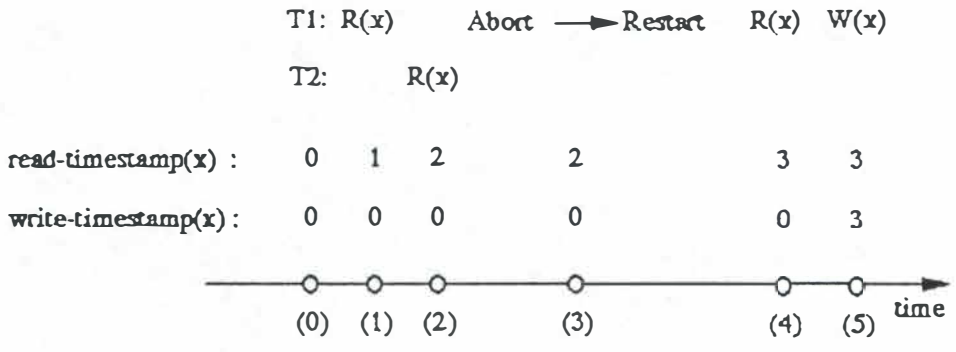
(2) $T2 : R(x)$ *is granted immediately, since* $T2$*'s timestamp is still larger than the write-timestamp of* x *which is* 1 *now.*

(3) *Livelock occurs to* $T1$ *at this stage as the request* $T1 : W(x)$ *is delayed forever. This is because the timestamp of* $T1$ *is smaller than the read-timestamp of* x *which is now* 2. *This will also block the execution of* $T1$ *indefinitely since read-timestamp of* x *will never get smaller than* 2.

In dynamic Two-Phase Locking mechanism, livelock is resolved by maintaining the delayed requests on a FIFO queue and prevent the newly arrived request from overtaking the requests already on the queue (*see section 3.2.3*). The request that may get into livelock problem is known to the concurrency controller before the problem occurs. Disallowing the subsequent requests to overtake this

request will prevent the problem of livelock. However, such strategy is not appropriate in dynamic Timestamping mechanism as the occurrence of livelock can not be foreseen. Livelock occurs at the moment the request concerned arrives at the concurrency controller. The *overtaking request* (*the request from the younger conflicting transaction*) is granted before the arrival of the *overtaken request* (*the request from the older transaction*); hence, there is no way to prevent the overtaking by maintaining a FIFO queue as the *overtaking request* has been granted well beforehand. In the other words, it is pointless to delay a request at all, since there is no chance that the delayed request will be granted in later stage. In such a situation, the only solution is to abort the transaction, undo all the updates and restart the transaction. The restarted transaction will be treated as a new transaction and a larger timestamp is given. Notice that this strategy is in fact a *pre-emptive strategy* described in *section 3.2.2.* The following example demonstrates the use of this strategy to resolve the livelock in previous example.

Example.



(1) $T1 : R(x)$ *is granted immediately.*

(2) $T2 : R(x)$ *is granted immediately.*

(3) $T1 : W(x)$ *is denied since the timestamp of* $T3$ *is smaller than the read-timestamp of* x. *Therefore,* $T3$ *is aborted and restarted with a new timestamp* 3.

(4) $T1 : R(x)$ *is granted immediately since the timestamp of* $T1$ *is larger than the read-timestamp of* x.

(5) $T1 : W(x)$ *is granted immediately as timestamp of* $T1$ *is larger than both the read-timestamp and write-timestamp of* x.

Therefore, the output schedule becomes $T2T1$.

3.3.3 Reading of Phantom Object

One major drawback of the Timestamping mechanism (both pre-transaction and dynamic) is that it does not take the second concurrency control problem - *reading of phantom object* - into consideration. In a Two-Phase Locking mechanism, this problem is prevented by locking the updated data object and delaying the subsequent requests to read this data object until the updating transaction unlocks the data object at the end of the transaction. However, similar constrainst is not imposed in Timestamping mechanism.

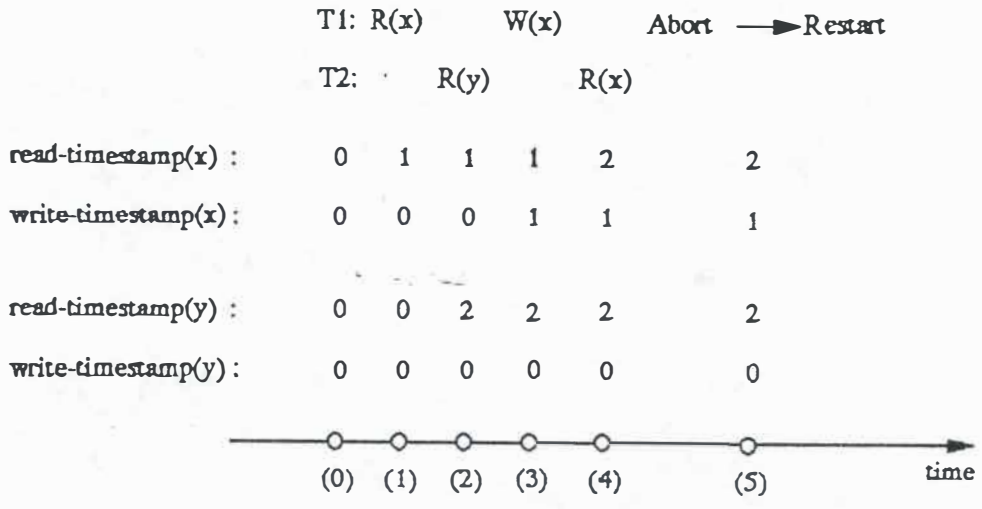
In dynamic Timestamping mechanism, a request to read a data object is granted as long as the timestamp of the transaction from which the request is originated is larger than the write-timestamp of the object. In pre-transaction timestamping mechanism, a read request is granted once all the write requests on the same data object from the older transactions have been granted. In the other words, a transaction is allowed to read a data object which has been updated by another transaction that is not yet terminated. The copy of the data object read may become a *phantom object* if the updating transaction is aborted later due to external factors (*see section 2.2.2*) or to resolve the livelock. The following example shows an occurrence of such problem.

Example. *Consider that the following schedule is submitted to the concurrency controller based on the dynamic Timestamping mechanism.*

T1 :	$R(x)$		$W(x)$		$W(y)$
T2 :		$R(y)$		$R(x)$	

The scheduling of this input schedule by the concurrency controller is discribed below. (Assume that the read-timestamp and write-timestamp of all the data

objects are 0 *initially and transaction T1 and T2 are given the timestamps* 1 *and* 2 *respectively.*)



(1) $T1 : R(x)$ *is granted immediately since timestamp of* $T1$ *is larger than write-timestamp of* x.

(2) $T2 : R(y)$ *is granted immediately since timestamp of* $T2$ *is larger than write-timestamp of* y.

(3) $T1 : W(x)$ *is granted immediately since timestamp of* $T1$ *is equal to the read-timestamp and write-timestamp of* x.

(4) $T2 : R(x)$ *is granted immediately since timestamp of* $T2$ *is still larger than the write-timestamp of* x.

(5) $T1 : W(y)$ *is denied since timestamp of* $T1$ *is smaller than the read-timestamp of* y. $T1$ *is aborted and restarted with a new timestamp* 3. *During the abortion, write access* $T1 : W(x)$ *performed at stage*(3) *is undone. Hence, transaction* $T2$ *has read a phantom data object at stage*(4).

This problem is especially serious where dynamic Timestamping mechanism is used since the rate of abortion is likely to be much higher compared with the use of pre-transaction Timestamping mechanism. For this reason, dynamic Timestamping mechanism is less popular despite its simplicity. On the other hand, pre-transaction Timestamping mechanism is already in use in **SDD-1**[BERN80].

3.4 Optimistic Mechanism

Optimistic concurrency control mechanism is another *non-locking* mechanism. This mechanism was first introduced by H.T. Kung and J.T. Robinson [KUNG81]. It adopts an entirely different approach from the Two-Phase Locking and Timstamping mechanisms earlier described.

Concurrency controllers based on this mechanism *optimistically* assume that conflicts will not occur during the execution of the transactions and therefore allows all submitted requests to proceed without any delay and scheduling. The assumption will only be challenged at the termination of each transaction. Transactions that fail this validation stage are aborted and restarted from the beginning again.

Based on this strategy, degree of concurrency is much higher than the other two mechanisms if the chances of conflicts among transactions are low. Due to this feature, optimistic approach is more appropriate for application domains where queries are dominating, such as *Library Cataloguing DataBase Systems.*

3.4.1 The Three Phases of a Transaction

For a practical realization of the optimistic approach, [KUNG81] partitioned the execution of a transaction into three different phases, namely *Read Phase, Validation Phase* and *Write Phase*. (*See figure 3.14*)

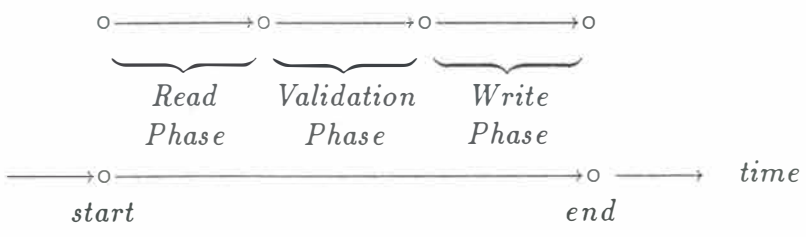


Figure 3.14 Three phases of a transaction

(1) **Read Phase** : In this phase, all requests submitted to the concurrency controller are granted immediately without any delay. To avoid unnecessary overhead in undoing the updates while the transaction is aborted later, only the requests to read the data objects that are not updated by the same transaction are directed to the data manager. All write requests are held at the concurrency controller until the transaction is validated to be *conflict-free*. At the mean time, a copy of the updated data object is stored in a local write set maintained for this transaction. Requests to read a data object which has been updated by the same transaction are granted with a local copy of the updated data object returned to the users. Of course, all these operations are invisible to the users. To them, transactions are carried out as if all read and write requests are directed to the data manager. Read phase is completed when the last request of the transaction is granted.

(2) **Write Phase** : *Write Phase* is the final stage of the execution of a transaction. Transactions that reach this phase are those which have been proved to be *conflict-free*. All write requests held at the concurrency controller that belong to these transactions are then released to the data manager which physically performs the updates on the data base.

(3) **Validation Phase** : This is the most important phase in the execution of a transaction. The assumption that no conflict will occur in the *read phase* is validated in this phase. Various methods of validation can be used here. For example, Papadimitriou [PAPA86] validates the transaction by maintaining a conflict graph; H.T. Kung [KUNG81] maintains a local read set and local write set for each transaction and check if these two intersect the *conflicting sets* [13] belong to the older transactions. Despite these different approaches, the common objective of all these validation methodologies is to ensure the *conflict serializability* of the output schedule. Papadimitriou's method in validating a transaction is more powerful than Kung's method in

[13] *Two local sets are considered as conflicting sets if at least one of them is a local write set.*

the sense that the former allows conflict to occur between two transactions and abort only when deadlock is detected in the conflict graph. Therefore, aborting and restarting transactions in the concurrency controllers that adopt Papadimitriou's method are not as frequent as in those that apply Kung's method. However, this method also introduces the overhead that non-locking mechanisms are trying to avoid, that is the overhead in *deadlock detection*. As Kung's methodology is rather lengthy, we refer to [KUNG81] for a more detailed discussion.

3.4.2 Deadlock and Livelock

Problem of deadlock does not exist in Optimistic mechanism. Since all transactions are allowed to proceed without being delayed, no transaction is waiting for another to release the data objects as in the Two-Phase Locking mechanism; hence, problem of deadlock will not arise at all. Similarly, as there is no delaying and waiting, there is no such problem of indefinitely delay - *livelock*. However, Optimistic mechanism faces another form of livelock problem, that is **repeatedly restart.**

Repeatedly restarting a transaction is very common, especially when the transaction is accessing large number of data objects. It increases the chances of the occurrence of conflicts and so the need for abortion. In [KUNG81], such problem is resolved by keeping track of the number of times a transaction fails the validation. The transaction with large number of failures is allowed to proceed with all other younger transactions *hold-up* outside the concurrency controller until this transaction terminates.

Chapter 4
Implementation of Concurrency Control Mechanisms

4.1 Introduction

In the previous chapter, we introduced the basic features of the three different concurrency control mechanisms. Designing and developing algorithms to implement these mechanisms and strategies is the major objective of this chapter.

A '*colours*' file is used as a sample database to demonstrate how the concurrency control mechanism correctly schedules the accesses to this file. This indexed file consists of a list of people's names and their favourite colours. Each record maintained in the file has the following data structure :

```
colour_record = record
                  name : [KEY(0)]string;
                  favourite : colour;
                end;
```

where name is the indexed field.

An overview of the sample database system is depicted in *figure 4.1*.

There are all together five different modules which are

1. **Concurrency Controller** -

This is the major module in the system. Two concurrency control mechanisms, namely *Two-Phase Locking mechanism* and *Timestamping mechanism*[14], discussed in last chapter will be used to implement this module.

2. **Record Inserting Program Module** -

This module responsible for inserting new records into the sample '*colours*' database.

3. **Colours Database** -

A simple database consists of a list of people's names and their favourite colours.

[14] *Implemetation issues of Optimistic mechanism is not discussed in this thesis.*

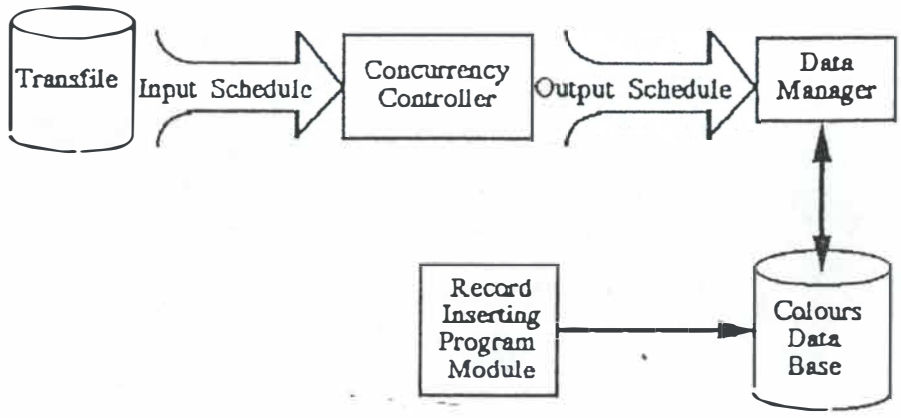


Figure 4.1 Overview of the sample database system

4. **Data Manager** -

Module responsible for physical interaction with the *Colours Database*.

5. **Transfile** -

A sequential file consists of a sequence of requests. It is used to simulate an infinite stream of requests generated from different sites[15]. Formats of the request will be described separately in the discussions of the implementations of various concurrency control mechanisms later.

Note that the major purpose of building this simple incomplete database system is to provide an environment to demonstrate different concurrency control mechanisms discussed in last chapter. The major module in this system is the concurrency controller and the other modules are simply used as supporting modules to show the working of these mechanisms. Therefore, this chapter is focusing on *designing and developing different algorithms to implement the 'concurrency controller' module*. Other supporting modules are kept as simple as possible and are not discussed here.

[15] *In this thesis, transactions are assumed to be activated at the sites different from where the central DBMS is located, such as the banking systems described in section 2.2.*

Two concurrent programming techniques are applied to implement the algorithms. These techniques include synchronization by **semaphore** and **monitor**. As mentioned previously, direct application of these techniques in the implementations of concurrency control mechanisms may turn out impractical or may not work at all[BERN81]. Therefore, certain modifications are made in this chapter to make them applicable in this domain.

PASCAL has been chosen as the vehicle for this implementation. All data structures and algorithms will be presented in PASCAL or near PASCAL format throughout the chapter.

4.2 Two-Phase Locking

4.2.1 Semaphore

In practice, *synchronization by semaphore* is commonly used as a technique to implement *LOCK* and *UNLOCK* operations in Two-Phase Locking mechanism[PAPA82]. This technique was first introduced by Dijkstra[DIJK65]. Dijkstra's concept of altering the values of *semaphores* by two primitive operations *wait* (or P) and *signal* (or S) to synchronize concurrent accesses to a shared resource has been widely used in Operating Systems since then. In a Database Management System, this technique is probably the most appropriate one for implementing Two-Phase Locking mechanism. Of course, certain modifications are needed to make it suitable.

Generally, the primitive operations *wait* and *signal* are implemented as follows :

wait(s) : **if** $s > 0$
then *decrement* s *by* 1
else *hold this process till* $s > 0$

signal(s) : *increment* s *by* 1

where s *is a semaphore which will be set to* 1 *for (binary semaphore) or* N *(for counting semaphore) initially.*

A *First-In-First-Out* (FIFO) queue is normally maintained for each *semaphore* to hold the waiting processes. The first process on the queue will be awaken and

removed from the queue once the process possessing the shared resource performed the operation *signal*(*s*). Base on this approach, a general data structure of a *semaphore* is shown in *figure 4.2*.

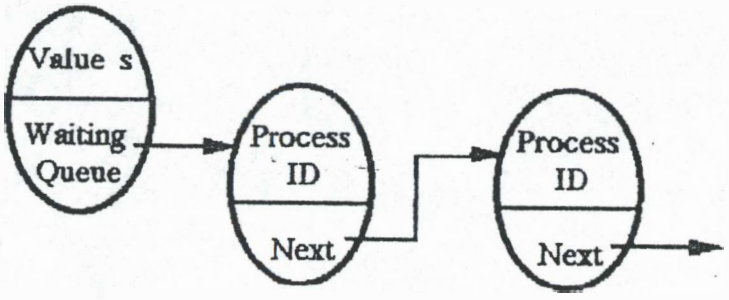


Figure 4.2 General data structure of a semaphore

The *wait* and *signal* operations are similar to the *Lock* and *Unlock* operations in Two-Phase Locking Mechanism. Each transaction's request can be thought as a process requesting a shared resource. If a transaction's request to access a shared data object fails the *Lock* (*wait*(*s*)) test, it will be held on the waiting queue associate with this data object; otherwise, the request will be granted and submitted to the data manager. *Unlock* (*signal*(*s*)) operation unlocks the data object and allows another request on the waiting queue associate with this data object to be granted. A "*lockbit*" can be allocated to each accessible data object as a *semaphore*. Unfortunately, the implementation of *Lock* and *Unlock* is not that simple. There are three basic differences between the *Lock/Unlock* and *wait/signal* operations. These differences are :

- There are two different types of *Lock* in Two-Phase Locking mechanism, namely *Read-Lock* and *Write-Lock*, each has a different function;
- *Lock* test is not as simple as the test carried out by *wait* operation. As described in *section 3.2*, a request to lock a data object will be denied if its lock conflicts with the lock/s currently held on that data object; otherwise, the request will be granted and sent to data manager;

- Not only the delayed requests are maintained on the waiting queue associate with the corresponding data object, copy of each granted request of the active transactions has to be made available to the concurrency controller too. This allows the concurrency controller to keep track of all transactions currently holding locks on the data object.

Due to the above differences, certain modifications are made to the data structure of the *lockbit* (*semaphore*) as well as the *lock* (*read and write*) and *unlock* operations.

4.2.2 Lockbit

As pointed out in last subsection, a copy of each granted request accessing a data object has to be available to the concurrency controller. This information is very important to the subsequent requests accessing the same data object. Therefore, another FIFO queue, which is called *granted queue* here, is maintained alongside with the waiting queue. This queue holds the syntactic information of all requests currently locking the corresponding data object. Removal of a granted request from the granted queue (due to the request to unlock, termination or abortion of the transaction from which the request is originated) may result in transferring one or more waiting request/s from the waiting queue to the granted queue. Each transfer signifies a request being granted.

Based on the above requirements, the data structure of the *lockbit* is revised as in *figure 4.3*.

Two additional pointers, *last_wait_request* and *last_granted_request*, pointing to the last requests of waiting queue and granted queue respectively may be added to the *lockbit*. This will allow easy addition of requests to both queues.

In PASCAL, such data structure are expressed as follows :

```
lockbit = record
            s : integer;                {s value}
            wait_que : lock_ptr;        {waiting queue}
            w_last : lock_ptr;          {last waiting request}
            granted_que : lock_ptr;     { granted queue}
            r_tail : lock_ptr;          { last granted request}
          end;
```

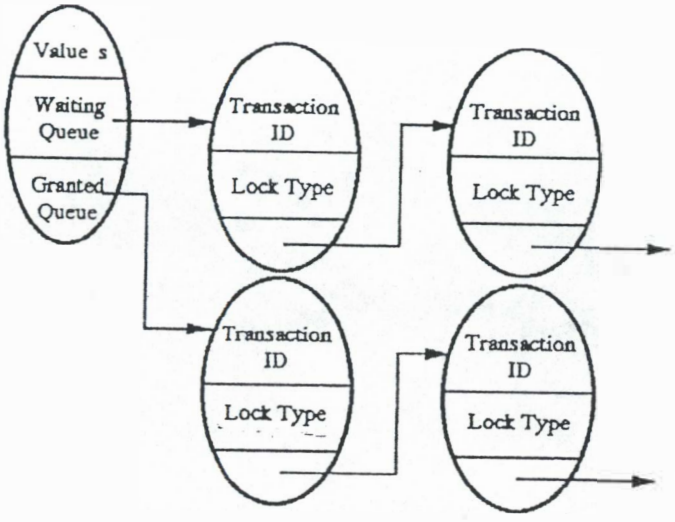


Figure 4.3 Data structure of a lockbit

where

```
lock_ptr  =  ↑ lock_list ;

lock_list =  record
               Tid    : integer;     {Transaction ID}
               Tlock  : lock_flag;   { Lock type, R or W}
               next   : lock_ptr;
             end;
```

Another difference of a *lockbit* from a normal *semaphore* is the change of the value s. Generally, both *binary semaphore* and *counting semaphore* do not allow *negative value*, but this restriction is ignored in this implementation.

In this implementation, the change of value s in each *lockbit* is according to the following rules :

1. When a data object is first added to the database, the value s of its *lockbit* is initialized to 1.
2. An arrival of a request to lock a data object will decrement the value of s by 1. (*This rule applies to both granted and delayed requests.*)
3. A request to unlock will increment the value of s by 1.
4. Value of 1 indicates that both waiting queue and granted queue are empty.

5. A *negative* or *zero* value for s indicates that there is at least one request on the waiting queue or granted queue.

The value of s associate with the *lockbit* is not as significant as that associate with the *semaphore*. Its value alone cannot decide the grant or denial of a request to lock a data object, but it plays a very important role in *semaphore* (*see section 4.2.1*). The $s = 1$ value in a *lockbit* serves as a flag to indicate the emptiness.

At this stage, single granularity of lock is assumed. Record in the '*colours*' database is the only *lockable* data object. A *lockbit* is allocated to each record maintained in the database.

Beside checking the value of s, thorough test is required on each request submitted to the concurrency controller to decide if that request can be granted or shall be delayed. Details about this test will be discussed in next subsection.

4.2.3 The Compatibility Test

In Two-Phase Locking mechanism, decision to grant or delay a request to lock a data object is entirely based on the compatibility of the requesting lock and the locks currently held at that data object. Testing of the compatibility of a requesting lock is carried out by the following algorithm :

ALGORITHM 4.1

Case 1 :

Supposing transaction T_i is requesting a Read Lock on data object O.

```
Function Compatible : Boolean ;
begin
  find_data_object(O) ;
  For each granted_request ∈ granted_queue do
    if  (i <> granted_request.Transaction_id) and
        (granted_request.lock_type = Write_Lock)
    then
        join_conflict_id_list(granted_request.Transaction_id) ;
  Compatible := (conflict_id_list = φ) ;
end;
```

Case 2 :

Supposing transaction T_i is requesting a Write Lock on data object O.

```
Function Compatible : Boolean ;
begin
  find_data_object(O) ;
  For each granted_request ∈ granted_queue do
    if  (i <> granted_request.Transaction_id)
    then
        join_conflict_id_list(granted_request.Transaction_id) ;
  Compatible := (conflict_id_list = φ) ;
end;
```

We can now make some remarks on the above algorithm :

1. *Conflict_id_list* is used to maintain the identifiers of the active transactions which are conflicting with the transaction T_i. (*Note that two transactions are in conflict if their locks on a same data object are incompatible. See section 3.2.1.*)
2. In case 1, incompatibility arises when at least one of the granted requests is a Write-Lock request submitted by a transaction T_j, where $T_i \neq T_j$.
3. In case 2, incompatibility arises when at least one of the granted requests is submitted by a transaction T_j, where $T_i \neq T_j$.
4. *Livelock* is likely to occur in the above algorithm. Note that only the requests on the granted queue (*requests currently holding locks*) are tested for the compatibility with the T_i's request. This allows the newly arrived request to join the granted queue and may then create more conflicts with the transactions whose requests are on the wiating queue. Such an *unfair privilege* may give rise to the problem of livelock. (*See section 3.2.3 for example*)

To resolve this livelock problem, additional test given in *section 3.2.3* is included. Now, the compatibility tests are carried out as follows :

Case 1 :

```
Function Compatible : Boolean ;
begin
  find_data_object(O) ;
  For each granted_request ∈ granted_queue do
    if  (i <> granted_request.Transaction_id) and
        (granted_request.lock_type = Write_Lock)
    then
        join_conflict_id_list(granted_request.Transaction_id)
    else
    if (i = granted_request.Transaction_id)
    then transaction_granted_before := True ;

  if transaction_granted_before or (granted_queue = φ)
  then Compatible := True
  else
  begin
    For each wait_request ∈ wait_queue do until
        ((wait_request.Transaction_id = i) or
          (all requests are done))
      if wait_request.lock_type = Write_Lock
      then
          join_conflict_id_list(wait_request.Transaction_id) ;
    Compatible := (conflict_list = φ) ;
  end ;
end ;
```

```
Case 2 :
    Function Compatible : Boolean ;
    begin
      find_data_object(O) ;
      For each granted_request ∈ granted_queue do
        if  (i <> granted_request.Transaction_id)
        then
            join_conflict_id_list(granted_request.Transaction_id)
        else
            transaction_granted_before := True ;

      if (not transaction_granted_before) or
        (conlfict_id_list <> ϕ) or
        (granted_queue = ϕ)
      then
        For each wait_request ∈ wait_queue do until
            ((wait_request.Transaction_id = i) or
              (all requests are done))
          join_conflict_id_list(wait_request.Transaction_id) ;

      Compatible :=  transaction_granted_before and
                     (conflict_list = ϕ) ;
    end ;
```

The above revised tests introduce the following new features in Algorithm 4.1 :

1. In case 1, if there is no granted request originated from T_i and there is a Write-Lock request on the wait queue, the Read-Lock submitted by T_i is considered as incompatible. This will disallow the Read-Lock request overtaking any request on the wait queue and join the granted queue.
2. In case 2, as long as there is a waiting request (Read or Write Lock) on the wait queue and there is no granted request originated from T_i, the Write-Lock request submitted by T_i is considered as incompatible. Again, this will prevent the Write-Lock from joining the granted queue.
3. Additional control condition,

   do until *wait_request.Transaction_id* $= i$

   allows Algorithm 4.1 to be used to re-check the compatibility of the waiting requests once a granted request is removed (ie. one of the locks has been released).

4.2.4 Dynamic Two-Phase Locking

Implementation of Dynamic Two-Phase Locking presented here is based on the following locking protocol (*see section 3.2.5*) :

> "*Lock the data object immediately before each request is made and unlock all the locked data objects after the last request is granted.*"

Before proceeding with the detailed discussion of the implementation of this locking protocol, format of the requests submitted to the concurrency controller and the strategy used to detect and resolve deadlock are described.

4.2.4.1 Request

Request to access a data object (*a 'colours' record in our sample database*) is represented in the following format :

$$< Transaction\ ID >< Access\ Type > [< AccessKey >]$$

where

$< Transaction\ ID >$ is the identifier of the site at which the transaction is originated;

$< Access\ Type >$ includes R (*read access*), W (*write access*) and E (*end of transaction*);

$< Access\ Key >$ is the primary key of the data object accessed by this request, which is *name* in our sample database;

(*Note* $< AccessKey >$ *is omitted if* $< AccessType >$ *is* E)

In PASCAL, the above request format can be represented as follows :

```
request = record
            id : integer;
            case
                access : lock_flag of
                         R, W : (access_key : string);
                         E    : ();
            end;
```

4.2.4.2 Deadlock Detection

As pointed out in *section 3.2.5*, the dynamic Two-Phase Locking mechanism faces the threat of deadlock. Detecting and resolving deadlock is a major problem in the implementation of this mechanism. Such an activity is expensive but unavoidable.

A **Conflict Graph** is used in this implementation to detect the occurrence of deadlock. The data structure of a node in this directed graph is shown in *figure 4.4*.

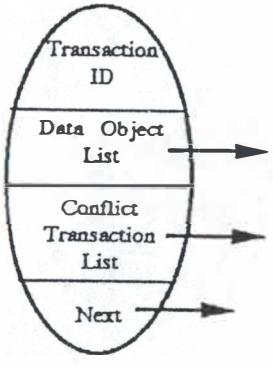


Figure 4.4 Data structure of a node of a Conflict Graph

The *Data Object List* maintains a list of data objects which have been requested by the transaction. The *Conflict Transaction List* is a list consists of pointers pointing to nodes corresponding to other transactions waiting for this transaction to release its locks on the data objects. In other words, the pointers maintained in the conflict transaction list act as *directed arcs* linking the transaction nodes in the conflict graph. For clarity, we shall call these pointers as **conflict graph arcs**. The *NEXT* field in the node maintains a pointer pointing to another node corresponding to another active transaction. Omitting the *Data Object List* and the *Conflict Transaction List*, the node is in fact a node of a FIFO queue maintaining all identifiers of the active transactions.

In PASCAL, this data structure can be represented as :

```
Node_ptr   =  ↑ Node;

Node       =  record
                id           :  integer;
                object_list  :  object_ptr;
                last_object  :  object_ptr;
                conflict_list:  conflict_ptr;
                last_conflict:  conflict_ptr;
                next         :  Node_ptr;
              end;
where
conflict_ptr  =  ↑ conflict_list;

conflict_list =  record
                   transaction  :  Node;
                   next         :  conflict_ptr;
                 end;

object_ptr    =  ↑ object;

object        =  record
                   access_key   :  string;
                   next         :  object_ptr;
                 end;
```

Therefore, a conflict graph can be defined as,

$$conflict\_graph : Node;$$

Given a request R_i submitted by transaction T_i, the following algorithm is used to construct the conflict graph.

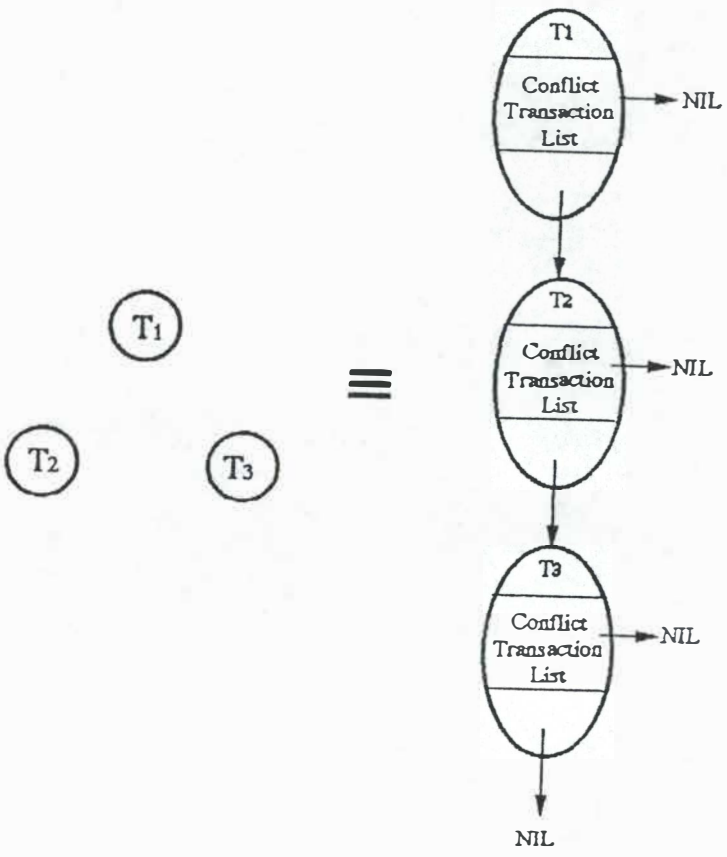
ALGORITHM 4.2

1. **If** a conflict graph is empty (*no active transaction*), create a new node for T_i and make this node a conflict graph; **Otherwise**, if a node that corresponds to T_i is not found in the conflict graph, create a new node for T_i and add this node to the graph by joining the FIFO queue; **Otherwise**, find the node which corresponds to T_i and add the data object requested by R_i to the *Data Object List* of that node.

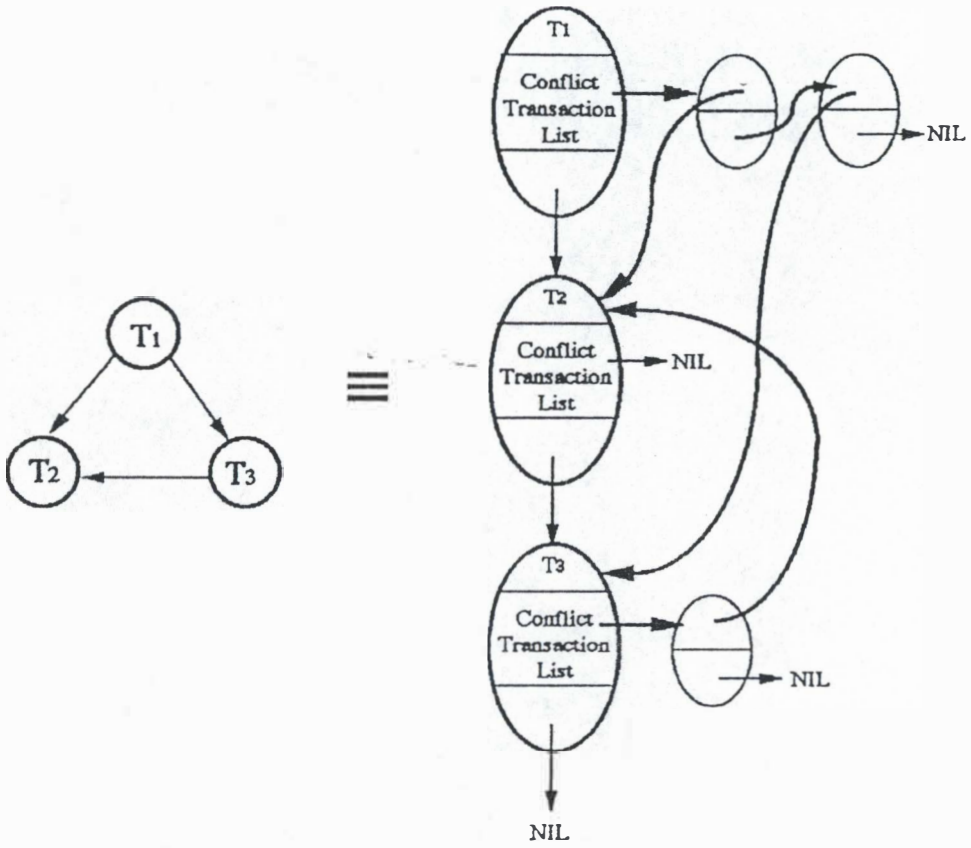
2. **If** the lock requested by R_i on a data object is incompatible with the lock/s currently held on that data object, T_i is conflicting with the transaction/s (say T_j) holding the lock/s, **then** create a new *conflict graph arc* from the *Conflict Transaction List* of the conflicting transaction (T_j) to T_i; **Otherwise**, do nothing and skip stage (3) below.
3. **If** a path can be found by travelling along the *conflict graph arcs* from the node corresponds to T_i back to itself , there is a **cycle** in the conflict graph and hence a *deadlock* has occurred.

The following three examples show three conflict graphs represented in the proposed data structure. (*Note that the Data Object Lists are omitted for the clarity of the graphs.*)

Example. *Three transactions without any conflict.*

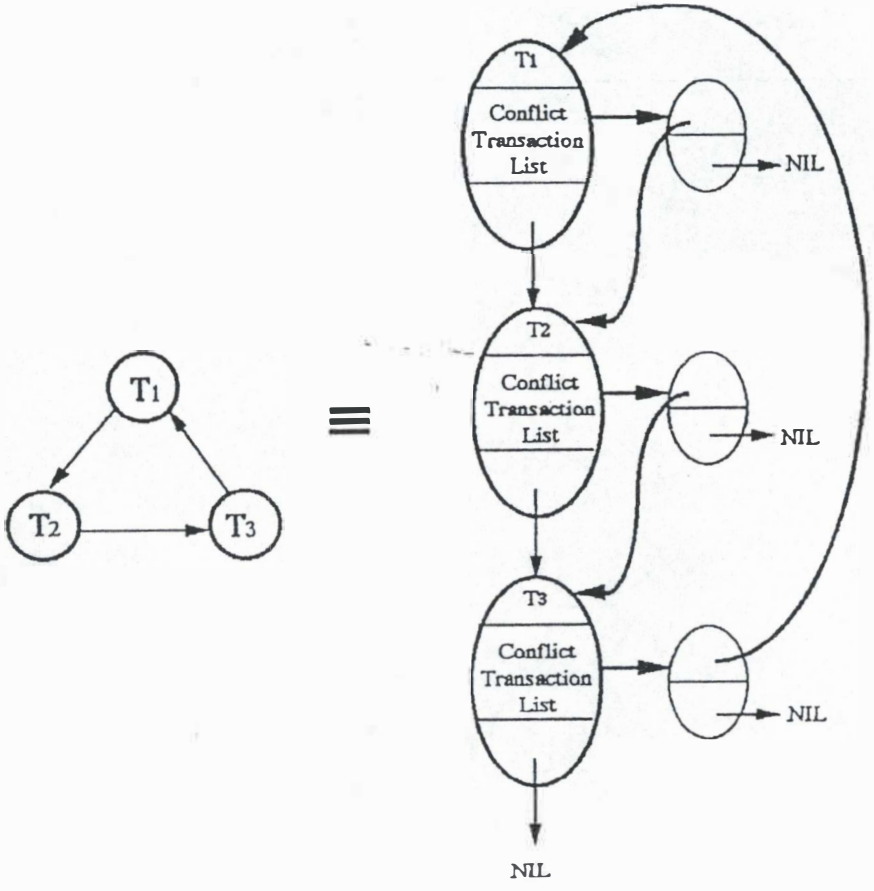


Example. *Three transactions with conflicts but no deadlock.*



(*Note that no path can be found by travelling from a node back to itself along the thicker lines.*)

Example. *Three conflicting transactions with deadlock.*



(Note there is a path from a node back to itself by travelling along the thicker lines.)

4.2.4.3 Implementation

(*Program based on the following algorithms is presented in Appendix A*)

Top Level Algorithm

The following algorithm is the top level algorithm of the dynamic Two-Phase Locking mechanism.

ALGORITHM 4.3

```
Repeat
  get_next_request(request) ;
  case request.access of
    R, W  :  begin
               update_conflict_graph(request) ;
               dblock(request) ;
             end ;
    E     :  end_transaction(request.id) ;
  end ;
Until eof(transfile) ;   { Until True; - in actual practice }
```

Algorithms of the three major procedures - *update_conflict_graph*, *dblock* and *end_transaction* - are given below.

Suppose R_i is a request originated from transaction T_i.

update_conflict_graph(R_i)

This procedure is an expansion of step one of Algorithm 4.2. In short, its task is to add a new transaction node for T_i into the conflict graph or update the object list of the transaction node that corresponds to T_i.

ALGORITHM 4.3.1

```
if conflict_graph = NIL then
  begin
    T_i_node := create_new_node(R_i) ;
    conflict_graph := T_i_node ;
    last_node := T_i_node ;
    {pointing at the most recently added node }
  end
else
  if not node_found(i,T_i_node) then
  {T_i is a new transaction}
    begin
      T_i_node := create_new_node(R_i) ;
      add_node_to_graph(T_i_node) ;
    end
  else
    { node corresponds to transaction T_i is found in the graph}
    add_object(R_i.access_key, T_i_node) ;
```

dblock(R_i)

This procedure attempts to lock the data object (say O) requested by R_i. It is the expansion of step two of Algorithm 4.2. Note that the compatibility test proposed earlier (*see Algorithm 4.1*) is used here. The algorithm is as follows :

ALGORITHM 4.3.2

```
O := find_data_object(R_i.access_key) ;
if O.lockbit.s > 0 then
  begin
    send_to_data_manager(R_i) ;
    join_queue(R_i, O.lockbit.granted_queue) ;
    O.lockbit.s := O.lockbit.s - 1 ;
  end
else
    begin
      if compatible(R_i, conflict_id_list) then
        begin
          send_to_data_manager(R_i) ;
          if no_other_granted_request(i) then
            { R_i is the first granted request from T_i }
            begin
              join_queue(R_i, O.lockbit.granted_queue) ;
              O.lockbit.s := O.lockbit.s - 1 ;
            end
```

```
        else
          begin
            previous_request := find(i, O.lockbit.granted_queue) ;
            if (R_i.access = W) then
                previous_request.access := W ;
          end
      end
    else  { not compatible }
      begin
        join_queue(R_i, O.lockbit.wait_queue) ;
        O.lockbit.s := O.lockbit.s - 1 ;
        add_arc_to_conflict_graph(R_i.id, conflict_id_list) ;
      end
  end ;
```

In the above algorithm, the procedure *add_arc_to_conflict_graph* is the most important procedure in the implementation of the Dynamic Two-Phase Locking mechanism. It consists of a deadlock detection mechanism which checks for the occurrence of *cycle* in the conflict graph each time a conflict arc is added. Algorithm of this procedure is shown in ALGORITHM 4.3.3.

ALGORITHM 4.3.3

add_arc_to_conflict_graph(*Transaction_id, conflict_id_list*)

```
if node_found(Transaction_id, T_i_node) then
For each conflict_id ∈ conflict_id_list do
  For each node ∈ conflict_graph do
    if node.id = conflict_id then
      begin
        { create new conflict arc }
        NEW(conflict_arc) ;
        conflict_arc ↑ .transaction := T_i_node ;
        conflict_arc ↑ .next := NIL ;
        if node ↑ .conflict_list = NIL then
          node ↑ .conflict_list := conflict_arc ;
        node ↑ .last_conflict := conflict_arc ;
        node ↑ .last_conflict := node ↑ .last_conflict ↑ .next ;
```

```
        if path(T_i_node, node) then
          { cycle detected - deadlock }
          begin
            remove_arc(conflict_arc, conflict_graph) ;
            end_transaction(i) ;
            abort_transaction(i) ;
          end ;
    end ;
```

Function *path* in the algorithm is used to find a path from the transaction node (T_i_node) to its conflicting transaction node (*node*). As the conflict arc is added into the conflict graph from node to T_i_node, a path found from T_i_node to *node* means that there is a *cycle* in the graph and therefore **deadlock** exists. *Path* is a recursive function which recursively calling itself to search through all possible paths extended from T_i_node. Algorithm 4.3.4 shows the algorithm of this recursive function. (*Note that this algorithm is the expansion of the third step of Algorithm 4.2.*)

ALGORITHM 4.3.4 : path(T_i_node, *node*)

```
if node ↑ .id = T_i_node ↑ .conflict_list ↑ .transaction ↑ .id then
  path := true      { cycle detected }
else
  For each transaction_node ∈
      T_i_node ↑ .conflict_list ↑ .transaction ↑ .conflict_list
    until (path is found) or (all transaction_node are done)
    if path(T_i_node, transaction_node) then
      path := true ;      { cycle detected }
```

end_transaction(*id*)

This procedure deletes the transaction node from the conflict graph, unlock all data objects locked by the transaction and inspects the new state of each waiting queue associate with the unlocked data objects. The algorithm of this procedure is shown in Algorithm 4.3.5.

ALGORITHM 4.3.5

```
if node_found(id, node) then
  begin
  remove_node(node) ;
  with node ↑ do
    For each object ∈ object_list do
      begin
        O := find_data_object(object.access_key) ;
        db_unlock(O, id) ;
      end ;
  end ;
```

Algorithm 4.3.6 and 4.3.7 given below present the algorithms of the two procedures *db_unlock* and *check_wait_request_state*. *db_unlock* unlocks all the data objects locked by the transaction. *check_wait_request_state* inspects the new state of each waiting queue associates with the unlocked objects by re-testing the compatibility of each waiting request. (*Notice that compatibility test given in Algorithm 4.1 is used again here.*)

ALGORITHM 4.3.6 : db_unlock (*Object, Transaction_id*)

```
with Object do
  begin
    For each granted_request ∈ lock_bit.granted_queue do
      if granted_request.id = Transaction_id then
        begin
          lock_bit.s := lock_bit.s + 1 ;
          remove_request(granted_request, lock_bit.granted_queue)
        end ;
        check_wait_request_state(lock_bit) ;
  end ;
```

ALGORITHM 4.3.7 : check_wait_request_state (*lock_bit*)

```
with lock_bit do
  For each wait_request ∈ wait_queue do
    if compatible(wait_request, conflict_id_list) then
      begin
        send_to_data_manager(wait_request) ;
        remove_request(wait_request, wait_queue) ;
        join_queue(wait_request, granted_queue) ;
      end ;
```

4.2.5 Pre-transaction Two-Phase Locking

The implementation of Pre-transaction Two-Phase Locking is simpler than Dynamic Two-Phase Locking. However, the class of the correct schedules that it can recognize is smaller too (*See section 3.2.5*). As discussed in *section 3.2.4*, the locking protocol of this mechanism is

> "*Lock all the data objects accessed by the transaction prior to the arrival of the first request of that transaction and unlock all the data objects after the last request is granted.*"

Before the detailed discussion of the implementation of the above protocol, a revised format of the requests submitted to the concurrency controller and the data structure of a linear linked list used to maintain the information about the active transactions are described.

4.2.5.1 Request

In order to cater for the lock request submitted prior to the submission of the transaction, the format of the request given in *section 4.2.4.1* is revised as follows

:

$$< Transaction\ ID >< Access\ Type > \\ [< Access\ Key > \mid \\ < Object\ Locks > \{< Object\ Locks >\}]$$

$$< Object\ Locks >:== \\ < Lock\ Type >< Object\ Count >< Access\ Key > \{< Access\ Key >\}$$

where

$< Access\ Type >$ includes R (*read access*), W (*write access*), E (*end of transaction*) and L (*lock request*);

$< Object\ Locks >$ applied only when $< Access\ Type >$ is L;

$< Lock\ Type >$ includes R (*read lock*) and W (*write lock*).

$< Object\ Count >$ indicates the number of data objects accessed with access type specified in the parameter $< Lock\ Type >$ preceded it;

```
1 L R 1 john W 2 jenny jerry
1 R john
2 L W 1 jim
1 W jenny
2 W jim
1 W jerry
2 E
1 E
```

Figure 4.5 Examples of request

(*Figure 4.5* shows a few examples of the request.)

Notice that the requests to lock the data objects are done in one single atomic action. This is to prevent the interleaving of the lock requests from different transactions. Allowing interleave of lock requests will cause the problem of deadlock which the pre-transaction Two-Phase Locking mechanism is avoiding.

4.2.5.2 Active Transaction List

Since deadlock does not exist in this mechanism, it is unnecessary to maintain a conflict graph and a deadlock detection mechanism. In the implementation of the dynamic Two-Phase Locking mechanism, information about an active transaction are maintained in a conflict graph. Without the conflict graph, these information are maintained in a linear linked list which is called *transaction_list* in this implementation. The data structure of a node in this linked list is shown in *figure 4.6*.

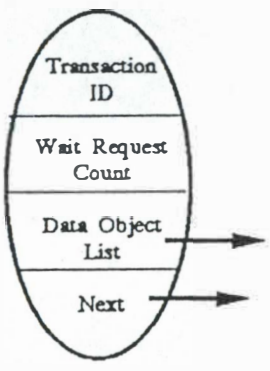


Figure 4.6 Data structure of a node in the transaction list.

In PASCAL, this data structure can be represented as :

```
active_ptr = ↑ active_transaction ;
active_transaction = record
                       id : integer ;
                       wait_request_number : integer ;
                       object_list : object_ptr ;
                       last_object : object_ptr ;
                       next : active_ptr ;
                     end ;
```

where

wait_request_number indicates the number of lock requests which have not been granted.

4.2.5.3 Implementation

(*Program based on the following algorithms is presented in Appendix B*)

Top Level Algorithm

The algorithm of Pre-Transaction Two-Phase Locking mechanism is rather simple. Algorithm 4.4 shows the top level algorithm of this mechanism.

ALGORITHM 4.4

```
Repeat
  get_next_request(request) ;
  case request.access of
    L    : new_transaction(request) ;
    R, W : send_to_data_manager(request) ;
    E    : end_transaction(request.id) ;
  end ;
Until eof(transfile) ;   { Until True; - in actual practice }
```

The two major procedures in this algorithm are *new_transaction* and *end_transaction*. The simplicity of this algorithm is that each *read* or *write* request arrived at the concurrency controller is sent to the data manager immediately. No compatibility test is required upon the arrival of each request as all the tests have been done in *new_transaction* when the lock request is submitted.

ALGORITHM 4.4.1 : new_transaction (*request*)

```
{ create a new transaction node }
NEW(node) ;
with node↑ do
  begin
    id := request.id ;
    wait_request_number := 0 ;
    object_list := NIL ;
    last_object := NIL ;
    next := NIL ;

    { attempts to lock all data objects accessed }
    lock_access_object(request, object_list,
                       last_object, wait_request_number) ;
  end ;

{ add new transaction node to the active transaction list }
add_node_to_list(transaction_list) ;
```

In the above algorithm, the procedure *lock_access_object* attempts to lock all data objects specified in the lock request. If all locks are granted, the site at which the transaction originated is signaled to send the rest of the transaction; otherwise, the transaction is delayed until all locks are granted. Algorithm 4.4.2 illustrates the algorithm of this procedure.

ALGORITHM 4.4.2
lock_access_object
(*request, object_list, last_object, wait_request_number*)

```
For each request.object_lock do
  For each request.object_lock.access_key do
    begin
      add_object(request.object_lock.access_key, object_list) ;
      O := find_data_object(request.object_lock.access_key) ;
      dblock(O, request, request.object_lock.access_key,
          wait_request_number) ;
    end ;
if wait_request_number = 0
then signal_all_lock_granted(request.id) ;
```

The procedure *dblock* is slightly different from the *dblock* procedure presented in Algorithm 4.3.2. The changes are made to cope with the differences between the two different locking protocols. A revised algorithm of this procedure is given in Algorithm 4.4.3.

ALGORITHM 4.4.3 :

```
dblock(Object, request, access_key, wait_request_number)
    with Object do
      begin
        if lock_bit.s > 0 then
          begin
            join_queue(request, lock_bit.granted_queue) ;
            lock_bit.s := lock_bit.s − 1 ;
          end
        else
          if compatible(request, conflict_id_list) then
            { compatible }
            if not in_granted_queue(request.id) then
              begin
                join_queue(request, lock_bit.granted_queue) ;
                lock_bit.s := lock_bit.s − 1 ;
              end
            else  { not compatible }
              begin
                join_queue(request, lock_bit.wait_queue) ;
                lock_bit.s := lock_bit.s − 1 ;
                wait_request_number := wait_request_number + 1 ;
              end ;
      end ;
```

ALGORITHM 4.4.4 : end_transaction (*Transaction_id*)

```
    remove_node(Transaction_id, node) ;
      { remove node from the transaction list }
      with node ↑ do
        For each object ∈ object_list do
          begin
            O := find_data_object(object.access_key) ;
            db_unlock(O, Transaction_id) ;
          end ;
```

Notice that Algorithm 4.4.4 is similar to Algorithm 4.3.5, except that the node is removed from the transaction list instead of a conflict graph.

The procedure *db_unlock* releases all the locks held on the data object (O) by the transaction (*Transaction_id*) and checks if any of the lock requests on the waiting queue can be granted due to the release of locks. The algorithm of this procedure is similar to Algorithm 4.3.6 and therefore not repeated here. However, the procedure *check_wait_request_state* is slightly different. The revised algorithm

of this procedure is given in Algorithm 4.4.4.1.

ALGORITHM 4.4.4.1 : check_wait_request_state (*lock_bit*)

```
with lock_bit do
  For each wait_request ∈ wait_queue do
    if compatible(wait_request, conflict_id_list) then
      begin
        remove_request(wait_request, wait_queue);
        join_queue(wait_request, granted_queue) ;
        if node_found(wait_request.id, node) then
          node ↑ .wait_request_number :=
                node ↑ .wait_request_number − 1 ;
        if node ↑ .wait_request_number = 0 then
          signal_all_lock_granted(wait_request.id) ;
      end ;
```

4.2.6 Multiply-Granular Lock

In *section 3.2.6*, we mentioned that a concurrency controller with single granularity of lock is normally inefficient in practice. To improve its efficiency, **Multiply-Granular Lock scheme** is introduced. In [GRAY77], such locking scheme is supported by *hierarchical lock* strategy.

Based on the hierarchical lock strategy, the whole database is organized in a hierarchy, such as a tree. Each level of the tree corresponds to a granularity of lock. To lock a data object in the tree, all nodes along the path from the root to the node corresponds to the object must be locked with appropriate intention locks. Failure in getting lock on any of these nodes will delay the request at that level until the lock is granted. Grant and denial of lock on a node is based on the compatibility of the requesting lock and the existing locks held at that node. (*See figure 3.13 for the compatibility matrix.*)

To implement this type of locking strategy, concurrent programming technique based on *monitor* can be used. In the following subsections, definition of *monitor*, reasons of using *monitor* and details of incorporating the locking strategy in dynamic Two-Phase Locking mechanism will be given.

4.2.6.1 Monitor

The concept of *monitor* was first conceived by Dijkstra [DIJK71] and later refined by Brinch Hansen[BRIN73] and Hoare[HOAR74]. It was initially proposed for resolving synchronization problems occured in Operating Systems, but later proved to be very valuable for other applications which suffer from the similar problems.

A *monitor* is a programming construct that consists of a set of variables and procedures managing a shared hardware or software resource. A process attempts to access a shared resource has to invoke one of the procedures contained in the corresponding *monitor*. One restriction of the invocation is that no other processes are executing any of the procedures residing in the same *monitor* while the invocation is made. By enforcing this restriction, mutual exclusion of the access to the shared resource is gauranteed and hence various sychronization problems are avoided.

One may view a *monitor* as a closed enclosure with only a single entrance and exit. A process is not allowed to enter the *monitor* (by invoking a procedure) if there is another process in it. Prrocesses which are disallowed to enter a *monitor* will be blocked outside the *monitor* until the *monitor* is released. A FIFO queue is normally maintained for the delayed processes. The first process on the queue will be immediately awakened and enter the *monitor* once the process in the *monitor* has left. Again, certain synchronization primitives are required to '*delay*' (or *wait*) a process and '*signal*' delayed process to proceed.

Monitors gaurantee the mutually exclusive access to the shared resources, but may result in unnecessary delaying and additional overhead in queueing the processes. The rather rigid restriction described above is much appropriate for *monitors* administer the hardware resources, such as printers, which allow only one process at a time; but it also proved to be very inefficient to certain resources, such as database. The best example to illustrate the rigidity of the restriction is the *Readers and Writers problem*. This problem was first introduced by Courtois, Heymans and Parnas[COUR71] and solved by Hoare [HOAR74] by using *monitors*.

Based on the restriction mentioned above, not more than one reader is allowed to access a shared resource (say a database) simultaneously, even when their accesses will not change the invariant (*the database consistency*) of the corresponding *monitor*. Therefore, unnecessary delay and lower degree of concurrency are resulted. By relaxing the restriction, a solution to solve the *Readers and Writers problem* by using monitors was suggested by Hoare [HOAR74]. Based on Hoare's solution, more than one readers are allowed to access the database simultaneously, but a writer has the sole exclusive access to the entire database. Notice the similarity of this syncronization strategy and Two-Phase Locking protocol mentioned earlier.

4.2.6.2 Monitors and Hierarchical Lock Strategy

As mentioned earlier, the hierarchical lock strategy assumes that the database is organized as a hierarchical structure. Each level of the hierarchy corresponds to a class of resources and a granularity of lock. For example, consider a database which can be organized in a tree structure as depicted in *figure 4.7a*, its resource hierarchy will be as shown in *figure 4.7b*. In the rest of this subsection, this example will be used to illustrate the application of *monitor* in the implementation of the hierarchical lock strategy.

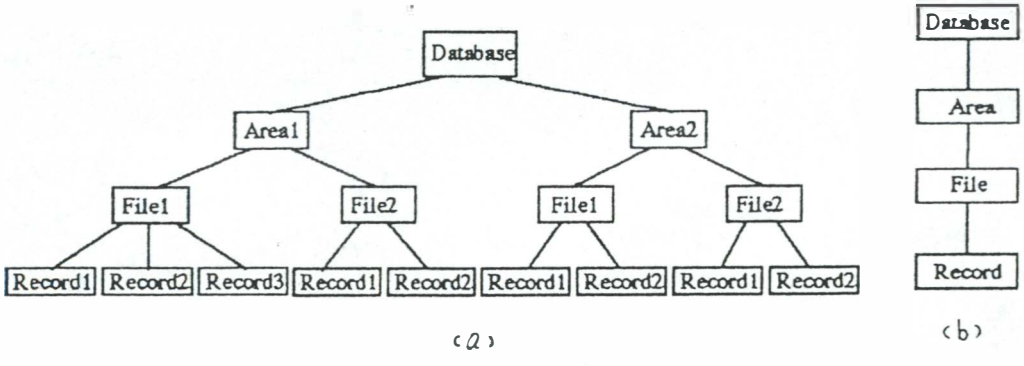


Figure 4.7 Data Base Hierarchy

Suppose there is only a single granularity of lock (*say a record*) and dynamic Two-Phase Locking mechanism is used to schedule the requests to access these records. One can consider each record is corresponding to a *monitor*, a *monitor* that consists of only a variable which maintains the record's state. A request to access a record will be delayed on the waiting queue associates with the corresponding *monitor*, if it fails the dynamic Two-Phase Locking protocol.

Next, suppose there is another granularity of lock which is *coarser* than a record, *say a file*. A *monitor* can be constructed to enclose all the variables and procedures (such as the conflict graph and deadlock detection mechanism) required to support the dynamic Two-Phase Locking mechanism on the records. A request to access a record has to acquire an appropriate intention lock on the file where the record is residing before it can enter the *monitor*. Requests that fail to acquire an intention lock on the appropriate file are blocked outside the *monitor* and delayed on the waiting queue associates with that file. We shall call this *monitor* as **record monitor**. Also, a request to read or write a file has to acquire an appropriate lock (*non-intention*) on this file before it can be granted. Once the lock is granted, the request can be submitted to the data manager immediately without entering the *record monitor*.

When another coarser granularity of lock (*say an area*) is considered, a similar *monitor* construct can be built by enclosing all the variables and procedures required to support Two-Phase Locking mechanism on the files into it. On addition, the *record monitor* described in last paragraph is included too. Like the *record monitor*, this **file monitor** has its own conflict graph and deadlock detection mechanism. The only difference is that the graph is built at the file level. All conflicts occur when the requests are attempting to acquire the lock (*intention or non-intention*) on the files will be added into this graph. Similarly, a request will not be allowed to enter the *file monitor* until it has acquired an appropriate intention lock on the area where the specified file is contained. A concurrency controller that support the hierarchical lock strategy on the database given in *figure 4.7* will have the *nested monitors* structure as shown in *figure 4.8*.

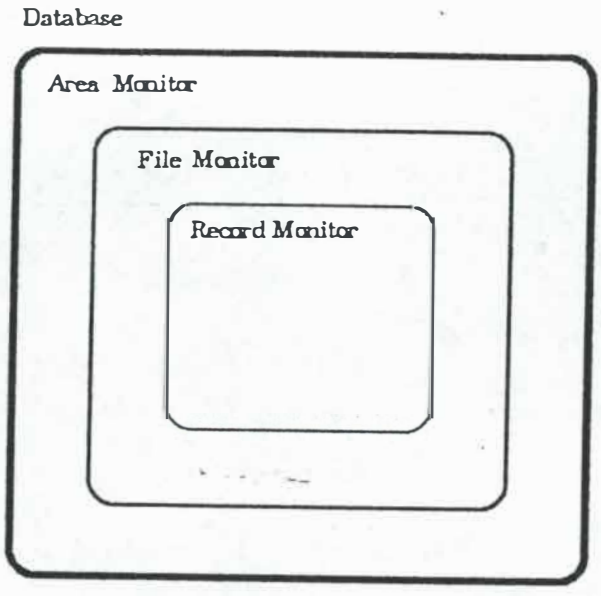


Figure 4.8 Nested Monitors

4.2.6.3 Implementation

As *monitor* construct is not offered in PASCAL, *procedure* will be used to simulate it. Instead of *nested monitors*, *nested procedures* are constructed to implement the hierarchical lock strategy. The program structure of the concurrency controller is shown in *figure 4.9*.

All *procedures* are identical. All of them have a conflict graph, deadlock detection procedure and other procedures required to support the Two-Phase Locking mechanism. The only difference is that they act at different levels of the database hierarchy.

In *section 4.2.4*, implementation of dynamic Two-Phase Locking mechanism was discussed. The implementation presented there assumed that the single granularity of lock is used. Suppose the granularity of lock is a *record* which is the finest grain in the database hierarchy given in *figure 4.7*, a *record monitor* can be constructed by enclosing all the procedures and variables developed in that subsection into a *procedure*.

To build a *coarser monitor*, say a *file monitor*, similar approach can be used. Instead of records, the accessible data objects are files. All procedures are per-

```
program Concurrency Controller ;
  procedure Scheuler(request) ;
    procedure Area_Monitor(request) ;
      procedure File_Monitor(request) ;
        procedure Record_Monitor(request) ;
          begin
            { Record_Monitor's Body}
          end ;
        begin
          { File_Monitor's Body }
        end ;
      begin
        { Area_Monitor's Body-}
      end ;
    begin
      { Scheduler's Body }
    end ;
  begin
    repeat
      Get_request(request) ;
      Scheduler(request) ;
    until True ;
  End.
```

Figure 4.9 Program structure of the Concurrency Controller.

formed at the file level and the file lockbits (*instead of record lockbit*) are administered.

To support a concurrency controller which allows two granularities of lock, *file* and *record*, *record monitor* is included into the *file monitor* as a sub-monitor. Of course, certain modifications have to be made to both monitors to support the hierarchical lock strategy. These modifications are :

1. **Request** -

A request submitted to the concurrency controller has to include both the identifier of the target record and the identifier of the file in which this record is residing. The granularity of lock (*file* or *record*) must be specified in each request as well. Hence, the following information must be supplied by each submitted request.

a. $< Transaction\ ID >$ - identifier of the site at which the transaction is originated;

b. $< Access\ Type >$ - includes R (*read access*) , W (*write access*) and E (*end of transaction*);

c. $< Granularity >$ - granularity of lock requested, either $File$ or $Record$.

d. $< File\ ID >$ - file identifier, such as *file name*;

e. $< RecordID >$ - record identifer, such as *record primary key*;

(*Note that information c,d and e are omitted if the* $< Access\ Type >$ *is* E.)

Two examples of the request are given below.

Example :

Suppose,

$< Access\ Type >$ $= R$
$< Granularity >$ $= File$,
$< File\ ID >$ $=$ '*student*',
$< Record\ ID >$ $=$ '*C.J.Date*'

which is a request to read a student record of the student named 'C.J. Date'. The granularity of lock suggests that the transaction from which this request is originated attempts to read all the records in the student file. Hence, if the read lock is granted or has been granted to this transaction in the file monitor, that implies that the transaction has gained a read lock on the student file and the request can be directed to data manager without locking the student record of 'C.J.Date'.

Example :

Suppose,

$< Access\ Type >$ $= W$
$< Granularity >$ $= Record$,
$< File\ ID >$ $=$ '*student*',
$< Record\ ID >$ $=$ '*C.J.Date*'

which is a request to update a student record of the student named 'C.J. Date'. The granularity of lock indicates that this request attempts to update this record only. To update this record, the request must first obtain a IW lock on the student file. Once the lock is granted, the request can then enter the record monitor and

request a write lock on the student record of 'C.J. Date'. The request will not be submitted to data manager till the write lock on the record is granted.

2. The Compatibility Testing Function

In the implementation of the dynamic Two-Phase Locking mechanism, a compatibility testing function was developed to evaluate the compatibility of a request. This function was designed to support the compatibility matrix which only consists of read lock and write lock (*see figure 3.3*). Obviously, this function can only be used in the *record monitor* but not other monitors support the *coarser grain locks*. Therefore, a revised version of the function is required to support the *multiply-granular lock compatibility matrix* (*see figure 3.13*). The algorithms given in *section 4.2.3* for handling requests requesting read lock and write lock are remained largely unchanged, except the conditions for a read request to join the conflict list required some minor modification. The new conditions are :

a. $(i <> granted\_request.Transaction\_id)$

and $(granted\_request.lock\_type\ in\ [IW, W])$

or

b. $wait\_request.lock\_type\ in\ [IW, W]$

The algorithms to handle the requests requesting intention locks (IR or IW) are similar to the algorithm handling read-lock requests. Again, the differences are their conditions to join the conflict list.

Case 1. The request is requesting an intention read lock (IR).

The required conflict conditions are :

a. $(i <> granted\_request.Transaction\_id)$

and $(granted\_request.lock\_type = W)$

or

b. $wait\_request.lock\_type = W$

Case 2. The request is requesting an intention write lock (IW).

The required conflict conditions are :

a. $(i <> granted\_request.Transaction\_id)$

and $(granted\_request.lock\_type\ in\ [R, W])$

or

b. $wait\_request.lock\_type\ in\ [R, W]$

3. **Granted Requests**

In the implementation of a dynamic Two-Phase Locking mechanism, a request is directed to the data manager once its lock on the data object is granted. The output strategy is slightly different here. If a request obtains an intention lock in a *monitor*, it will be submitted to the sub-monitor; otherwise, it will be sent to the data manager. In the other words, if the granularity of the lock obtained by the request is coarser than the grain size that it specifies, the request is submitted to the sub-monitor. As an example, if the grain size specified in a request is a record, this request will not be submitted to the data manager until a lock is granted in *record monitor*.

4.3 Timestamping

Even though the Timestamping mechanism is claimed as a non-locking mechanism, its strategy of excluding older conflicting transactions from accessing a data object is equivalent to a lock operation. Based on this argument, *semaphore* is used in the implementation of the Timestamping mechanism.

In the implementation of Two-Phase Locking mechanism, "*lockbit*" is allocated to each accessible data object as a *semaphore* (*see section 4.2.2*). In this section, another type of *semaphore* is introduced to implement the Timestamping mechanism. This *semaphore* is called "*timebit*". Similar to *lockbit*, *timebit* is assigned to each accessible data objects in the database. As the data structure and function of the *timebit* are different in pre-transaction and dynamic Timestamping mechanisms, they will be described separately when the implementations of the two mechanisms are discussed.

4.3.1 Pre-Transaction Timestamping Mechanism

As mentioned in *section 3.3.1*, pre-transaction Timestamping mechanism requires a transaction to provide the syntactic information, such as timestamp of the transaction, prior to the submission of the first request of the transaction. Based on these information, the concurrency controller can then perform the following scheduling protocol :

> *"A request to access an object is granted if all conflicting requests from the older transactions on the same object have been granted."*

To implement this protocol, an essential primitive construct, namely **timebit**, is required. Descriptions of this construct ,format of the request and a linked list maintaining the timestamps of the active transactions will be given below before the detailed implementation is discussed.

4.3.1.1 Request

In this implementation, request to access a data object is represented in the following format :

$< Transaction\ ID >< Request\ Type >< Access\ Key >,$
if $< Request\ Type > \in \{R, W\}$.

or

$< Transaction\ ID >< Request\ Type >< Syntactic\ Info >$
$\{< Syntactic\ Info >\}$
if $< Request\ Type >= P$.

$< Syntactic\ Info >::=$
$< Access\ Type >< Object\ Count >< Access\ Key > \{< Access\ Key >\}$

where

$< Request\ Type >$ includes R (*read request*), W (*write request*) and P (*pre-transaction information*);
$< Access\ Type >$ includes R (*read access*) and W (*write access*);
$< Object\ Count >$ indicates the number of data objects accessed with access type specified in the parameter $< Access\ Type >$ preceded it;

Notice that the format of the request is very similar to the format presented in the implementation of pre-transaction Two-Phase Locking mechanism (*see section 4.2.5.1*), except that notification of *end of transaction* is not required in this mechanism.

4.3.1.2 Timebit

In order to implement the pre-transaction Timestamping mechanism, two queues are allocated to each data object. These two queues are **read-timestamp queue** and **write-timestamp queue**. When the syntactic information of a transaction is provided, the timestamp of this transaction will be attached to one of these queue associate with the accessed data objects. For example, a transaction reading data object O will leave its timestamp (say t_i) on the read-timestamp queue associate with O prior to the first request of the transaction is submitted. Therefore, before a transaction is submitted to the concurrency controller, all data objects that this transaction is going to access should have its timestamp in the read-timestamp queue or write-stamp queue. A timestamp will not be removed from the queue until the following conditions are met.

(1) The corresponding request has arrived;

(2) There is no smaller timestamp on the write-timestamp queue;

(3) If this timestamp is on the write-timestamp queue, there must not have any smaller timestamp on the read-timestamp queue.

Notice that these three conditions are adequate to ensure the conflict serializability of the output schedule. *Figure 4.10* shows the data structure of a timebit. Each node maintained in the timestamp queue consists of three fields. These three fields are

(1) Transaction timestamp - an integer value;

(2) Transaction ID - an integer value;

(3) Arrived - a boolean value indicates the arrival of the corresponding request.

(*Note that timestamp is considered as an incremental integer value in this implementation.*)

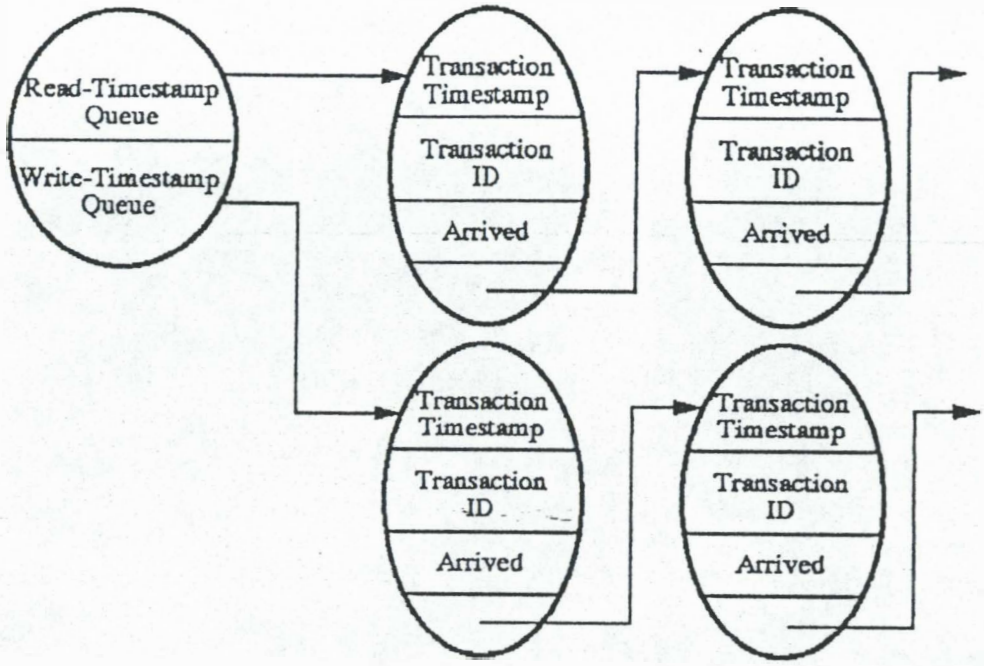


Figure 4.10 Data Structure of a timebit.

In PASCAL, the data structure of the timebit is expressed as follows :

```
timebit = record
            read_timestamp_queue : timestamp_ptr;
            last_read_timestamp  : timestamp_ptr;
            write_timestamp_queue : timestamp_ptr;
            last_write_timestamp : timestamp_ptr ;
          end ;
```

where

```
timestamp_ptr = ↑ time_stamp;
time_stamp    = record
                  stamp   : integer; {Transaction timestamp}
                  id      : integer; {Transaction ID}
                  arrived : boolean;
                  next    : timestamp_ptr;
                end ;
```

4.3.1.3 Active Transaction List

Similar construct was described in *section 4.2.5.2*. This linear linked list is used to maintain the syntactic information about all the active transactions. The information include the transaction identifier and the corresponding timestamp. The data structure of each transaction node maintained in this list is given in *figure 4.11*.

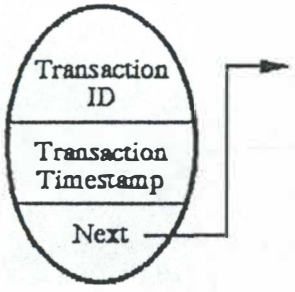


Figure 4.11 Data structure of a node of the transaction list

In PASCAL, this data structure is expressed as follows :

```
transaction_ptr  =  ↑ transaction_list;

transaction_list =  record
                      id : integer;
                      timestamp : integer;
                      next : transaction_ptr;
                    end ;
```

In the implementation, a pointer *last_transaction* is used to point at the last transaction on the list which is also the youngest transaction.

4.3.1.4 Implementation

(*Program based on the following algorithms is presented in Appendix D*)

Top Level Algorithm

The top level algorithm of the pre-transaction Timestamping mechanism is given in Algorithm 4.5.

ALGORITHM 4.5

```
repeat
  get_next_request(request);
  timestamp := transaction_timestamp(request.id);
  if request.access = P
  then claim_all_object(request, timestamp)
  else process_request(request, timestamp);
until eof(transfile);    { until True; - in actual practice }
```

In Algorithm 4.5, there are three major procedures, they are

1. *transaction_timestamp*(*id*) - a function that generates or returns a corresponding timestamp given the *id* of the transaction.
2. *claim_all_object*(*request*, *timestamp*) - a procedure which assigns transaction's timestamp to the timestamp queues (*read* or *write*) associate with each of the data objects requested by the transaction.
3. *process_request*(*request*, *timestamp*) - a procedure which grants or delays the requests submitted to the concurrency controller. Also, it is the most important procedure in the implementation of the pre-transaction Timestamping mechanism.

Detailed algorithms of these three procedures are given below.

ALGORITHM 4.5.1 : transaction_timestamp (*transaction_id*)

```
if transaction_list = NIL then
  {empty transaction list}
  begin
    {create new transaction node}
    New(node);
    with node ↑ do
      begin
        id := transaction_id;
        timestamp := 1;  {the oldest transaction}
        next := NIL;
      end ;

    join_transaction_list(node) ;
    transaction_timestamp := 1;  {RETURN}
  end
else  { transaction list not empty}
  begin
    For each transaction_node ∈ transaction_list do
      until (all transaction_node done) or (node_found)
        node_found := (transaction_node ↑ .id = transaction_id);
    if not node_found then
      { transaction node not found }
      begin
        { create new transaction node }
        New(node);
        with node ↑ do
          begin
            id := transaction_id;
            timestamp := last_transaction ↑ .timestamp + 1;
            next := NIL;
          end;
        transaction_timestamp := node ↑ .timestamp;  {RETURN}
      end
    else  { transaction node found }
      transaction_timestamp := transaction_node ↑ .timestamp;
      {RETURN}
  end;
```

ALGORITHM 4.5.2 : claim_all_object (*request, timestamp*)

```
For each request.syntactic_info do
  For each request.syntactic_info.access_key do
    begin
      O := find_data_object(request.syntactic_info.access_key);
      join_timestamp_queue(O, request.id, timestamp,
            request.syntactic_info.access_type);
    end;
```

The procedure *join_timestamp_queue* responsible for creating a timestamp node for the transaction and attaching this node to the appropriate timestamp queue of the specified data object.

ALGORITHM 4.5.3 : process_request (*request, timestamp*)

```
O := find_data_object(request.access_key);
case request.request_type of
  R  :  read_protocol(O, request, timestamp);
  W  :  write_protocol(O, request, timestamp);
end;
```

On the arrival of each request, the procedures *read_protocol* and *write_protocol* are used to decide if this request can be granted or has to be delayed. A request is granted only if its corresponding timestamp node maintained on the timestamp queue associate with the specified data object can be removed. Removal of a timestamp node depends on the three conditions given in *section 4.3.1.2*. In the other words, the request will not be granted if any one of these three conditions fails. On the other hand, a delayed request will update the $< Arrived >$ field in its corresponding timestamp node to *true* as an indication of its arrival.

After each removal of a timestamp node, another procedure *check_queue* which in turns recursively calling both *read_protocol* and *write_protocol* is applied to inspect each timestamp on both timestamp queues. If a timestamp is found to meet all the conditions mentioned earlier, it will be removed from the queue and its corresponding request will be granted. The procedures will be recursively proceed until all the timestamps on the read-timestamp queue and write-timestamp queue associate with the data object concerned are all checked. Algorithms of these three procedures are given below.

ALGORITHM 4.5.3.1 : read_protocol (*Object, request, timestamp*)

```
with Object do
  if (timebit.write_timestamp_queue = NIL) or
   (timestamp ≤ timebit.write_timestamp_queue ↑ .stamp)
  then
    begin
      send_to_data_manager(request);
      remove_time_stamp(timestamp, timebit, R);
      check_queue(Object, R);
    end
  else
    update_time_stamp(timestamp, timebit, R);
```

A request submitted to the procedure *read_protocol* is required to meet one of the following conditions before it can be granted.

1. *Object.timebit.write_timestamp_queue* is empty;
2. Its transaction timestamp is smaller than the timestamp of the first node on the *Object*'s *write_timestamp_queue*. It is unnecessary to check the rest of the queue for smaller timestamps. Note that timestamp nodes are attached to the queue in chronological order; therefore, if the transaction timestamp is smaller than the timestamp of the first node, it is smaller than the timestamps of the rest of the nodes on the queue. On the other hand, if the timestamp is larger than the timestamp of the first node on the queue, it has already failed the test and is pointless to check the rest.

There are two procedures in Algorithm 4.5.3.1 that required some brief descriptions. They are,

1. **remove_time_stamp**($t, timebit, access\_type$) - this procedure removes the timestamp node with timestamp t from the queue associate with *timebit*. *access_type* specifies which queue ($read-timestamp$ or $write-timestamp$) the node belongs to.
2. **update_time_stamp**($t, timebit, access\_type$) - this procedure updates the $<$ *Arrived* $>$ field in the timestamp node with timestamp t to *true*. This is to indicate the arrival of its corresponding request. *access_type* specifies which timestamp queue associate with *timebit* the node belongs to.

ALGORITHM 4.5.3.2 : write_protocol($Object, request, timestamp$)

```
with Object do
  if (timestamp ≤ timebit.write_timestamp_queue ↑ .stamp) and
    ((timebit.read_timestamp_queue = NIL) or
     (timestamp ≤ timebit.read_timestamp_queue ↑ .stamp))
  then
    begin
      send_to_data_manager(request);
      remove_time_stamp(timestamp, timebit, W);
      check_queue(Object, W);
    end
  else
    update_time_stamp(timestamp, timebit, W);
```

Similar to those requests submitted to *read_protocol*, a request submitted to *write_protocol* required to meet certain conditions before it can get granted. These conditions are listed below.

1. The timestamp of the transaction from which the request is originated is smaller than the timestamp of the first node on the *write_timestamp_queue*. Notice that this is similar to the second condition specified earlier in Algorithm 4.5.3.1.

and one of the following conditions,

2. *timebit.read_timestamp_queue* is empty.
3. The timestamp of the transaction from which the request is originated is smaller than the timestamp of the first node on the *read_timestamp_queue*.

ALGORITHM 4.5.3.3 : check_queue(*Object*, *access_type*)

```
with Object do
  begin
    if access_type = W then
    { A timestamp node has been removed from the write-timestamp
    queue. Check read-timestamp queue for nodes that can be removed.}
      For each timestamp_node ∈ timebit.read_timestamp_queue do
        if timestamp_node ↑ .arrived then
        {the corresponding request has arrived and being delayed}
          begin
          { re-create the request }
            request.transaction_id := timestamp_node ↑ .id;
            request.request_type := R;  {read request}
            request.access_key := Object.access_key;

            { re-submit the request to read_protocol }
            read_protocol(Object, request, timestamp_node ↑ .stamp);
          end;

      { compulsory check for both read and write access_type}
      For each timestamp_node ∈ timebit.write_timestamp_queue do
        if timestamp_node ↑ .arrived then
          begin
            { re-create request as above}

            { re-submit the request to write_protocol }
            write_protocol(Object, request, timestamp_node ↑ .stamp);
          end;
  end;
```

4.3.2 Dynamic Timestamping Mechanism

Implementation of dynamic Timestamping mechanism is probably the simplest among the mechanisms discussed in this thesis. A similar active transaction list is maintained and timebits are allocated to each accessible data objects as in the implementation of pre- transaction Timestamping mechanism. However, the data structure of timebit and the format of the request are much simpler than those described in *section 4.3.1*.

4.3.2.1 Request

Request in this implementation is represented in the following format :

$$< Transaction\ ID >< Request\ Type >< Access\ Key >$$

where

$< RequestType >$ includes R (*read request*) and W (*write request*).

4.3.2.2 timebit

Instead of maintaining two timestamp queues, timebit in this implementation requires to maintain only two timestamp values, namely **read-timestamp** and **write-timestamp.** (*Data structure of this timebit is shown in figure 4.12.*)

- *read-timestamp* - timestamp of the youngest transaction that last read the data object.
- *write_timestamp* - timestamp of the youngest transaction that last read the data object.

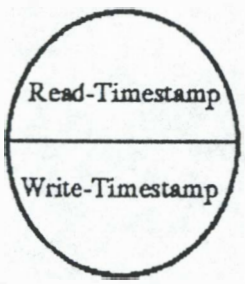


Figure 4.12 Data structure of a timebit.

In PASCAL, such data structure is expressed as follows :

```
timebit  =  record
              read_timestamp  :  integer;
              write_timestamp :  integer;
            end;
```

4.3.2.3 Implementation

Based on the scheduling protocol given in *section 3.3.2*, the following algorithms are developed.

(*Program based on the following algorithms is presented in Appendix C*)

Top Level Algorithm

The top level algorithm of the dynamic Timestamping mechanism is given in Algorithm 4.6.

ALGORITHM 4.6

```
repeat
  get_next_request(request);
  timestamp := transaction_timestamp(request.id);
  process_request(request, timestamp);
until eof(transfile)    { until True; - in actual practice }
```

Notice that Algorithm 4.6 is much simpler than the algorithm described in Algorithm 4.5. In this algorithm, procedure *transaction_timestamp* is a duplication of the procedure with the same name described in Algorithm 4.5.1 and hence will not be repeated here. However, there are some differences between the procedure *process_request* here and its counterpart described in Algorithm 4.5.3. The two sub-procedures *read_protocol* and *write_protocol* are so much simpler and easier to implement. Algorithms of these two sub- procedures are given in Algorithm 4.6.1 and 4.6.2.

ALGORITHM 4.6.1 : read_protocol(*Object, request, timestamp*)

```
with Object do
  if timestamp ≥ timebit.write_timestamp then
    begin
      send_to_data_manager(request);
      timebit.read_timestamp := timestamp;
    end
  else
    abort_transaction(request.transaction_id);
```

A read request submitted to procedure *read_protocol* will be granted if the timestamp of its transaction is larger or equal to the *write_timestamp* associates with the data object; otherwise, its transaction is aborted. The transaction which

successfully read the data object will replace its timestamp as the new read timestamp for that object.

ALGORITHM 4.6.2 : write_protocol(*Object*, *request*, *timestamp*)

```
with Object do
  if (timestamp ≥ timebit.write_timestamp) and
      (timestamp ≥ timebit.read_timestamp) then
    begin
      send_to_data_manager(request);
      timebit.write_timestamp := timestamp;
    end
  else
    abort_transaction(request.transaction_id);
```

A write request submitted to procedure *write_protocol* is required to meet more conditions to get granted. Beside the similar condition specified in Algorithm 4.6.1, the timestamp of the transaction from which the write request is originated has to be larger or equal to the *read_timestamp* associates with the data object as well; otherwise, the transaction is aborted. The transaction which successfully write the data object will replace its timestamp as the new write timestamp for that object.

Chapter 5
Conclusions and Problems in Distributed DBMS

In this concluding chapter, the objectives outlined in the introduction and the works presented in the last few chapters are summarized. Resistance posted by the concurrency control problems in distributed DBMS are discussed too.

5.1 Concluding Remarks

Concurrency control problems in centralized Data Base Management Systems have been well studied in the past two decades and a number of concurrency control mechanisms have been suggested with sound mathematical theory [PAPA86] as foundation. Among these mechanisms, three major approaches, namely **Two-Phase Locking Mechanism**, **Timestamping Mechanism** and **Optimistic Mechanism**, are surveyed. These three mechanisms differ in the degree of concurrency that they can achieve, the difficulty in implementation and other problems associated with each of them. Most reported works on these three mechanisms mainly focus on the theoretical aspects rather than techniques for implementing them. This thesis attempts to apply two major concurrent programming techniques - *synchronization and mutual exclusion by semaphore and monitor* - to the implementation of the Two-Phase Locking and Timestamping mechanisms.

Based on *Two-Phase Locking mechanism*, schedules output by the concurrency controllers are guaranteed to preserve the conflict serializability. This feature prevents the concurrent transactions from committing the concurrency control problems. However, it also introduces a most unwanted problem when dynamic information acquisition is applied, that is the problem of *deadlock*. Despite the advantage of being a deadlock-free mechanism, Pre-transaction Two-Phase Locking mechanism does not appear to be a better mechanism than Dynamic Two-Phase Locking mechanism due to its weaker scheduling criterion.

Synchronization by a *semaphore* is commonly used to allocate shared resources in Operating Systems. This is also a natural choice as a technique to implement a Two-Phase Locking Mechanism. Despite the similarity of the synchronization problems faced in both domains, this technique can not be applied in Data Base Management Systems without any modification. *Chapter 4* identifies this problem and suggests the necessary modifications to the semaphore and wait/signal operations to allow this synchronization technique to be used for the implementation of Two-Phase Locking mechanism.

Transactions accessing large subset of a data base are very common in most applications. Concurrency controllers based on Two-Phase Locking mechanism that support single granularity of lock are inefficient in handling this type of transaction. Due to this problem, *multiply-granular lock scheme* is supported by most existing DBMSs despite the additional lock management overhead imposed. The granularities of lock supported by several commercially available DBMSs are summarized in *figure 5.1*.

DBMS	Granularities of lock
IMS/VS	Segment type
	Segment instance
DMS1100	Area
	Page within area
System R	Segment
	Page
	Record

Figure 5.1 Granularities of lock [FERN81]

To incorporate multiply-granular lock scheme into Two-Phase Locking mechanism, synchronization by *semaphore* seems inadequate. A more structured approach based on *monitor* is suggested in *chapter 4*.

All schedules output by a concurrency controller based on Timestamping mechanism are conflict serializable to the serial execution of the transactions in chronological order. This important feature ensures the correctness of the output schedules and avoids the occurrence of deadlock. A proof was given in *chapter 3* to prove that deadlock is impossible in a concurrency controller based on this mechanism.

Although the *Timestamping mechanism* appears to be different from a locking mechanism, its scheduling strategy which excludes older transactions from accessing a data object already accessed by a younger transaction is equivalent to a lock operation. Based on this argument, *semaphore* is again used as a synchronization primitive in the implementation. Several modifications to the synchronization technique are suggested in *chapter 4* for its application to the Timestamping mechanism.

In *section 3.4*, another non-locking mechanism, *Optimistic mechanism*, was introduced. Concurrency controllers based on this mechanism optimistically assume that no conflict will occur among the concurrent transactions and allow all submitted requests from these transactions to proceed without any delay. This optimistic assumption will only be questioned at the end of each transaction. This type of scheduling strategy offers a high degree of concurrency in applications where queries are highly dominating; otherwise, it will introduce overhead in managing restarted transactions. This mechanism is discussed in *chapter 3*. Its implementation is not included in this thesis due to its entirely different approach.

In conclusion, this thesis surveys three major concurrency control mechanisms. It also suggests the use of concurrent programming techniques based on *semaphores* and *monitors* to implement two of the surveyed mechanisms, namely the Two-Phase Locking and the Timestamping mechanisms. Also, modifications to the suggested concurrent programming techniques were proposed to allow them to be used in the context of database concurrency control.

Future work should focus on the use of concurrent programming techniques based on **message passing**[WOOD87] and **data flow graph** in the implementation of concurrency control mechanisms. In particular, the use of **DBFG (Database Flow Graph)** [EICH88a,EICH88b] has attracted wide attention in the last few years. By using DBFG, scheduling in concurrency controller can be carried up in parallel. This in turns will also increase the degree of concurrency and transaction throughput. With the advantage of parallelism, this technique could also be applicable in parallel database machine environment in future.

5.2 Concurrency Control in Distributed DBMS

With the increasing number of users and their geographical dispersion, single site centralized DBMS has the following undesirable features :

1. **High communication cost** -

Complexity of the communication network connecting the central DBMS to the sites where the transactions are originated induces considerable costs. The costs incurred include the maintenance of the communication links and the CPU time required to send and receive the messages. For example, the American airlines TWA has to pay AT&T a million dollars of communications bill per month to maintain their airlines reservation network[GIFF84].

2. **Long communication delay** -

Access delay time is very much depend on the topology of the network used. Typical delay time is approximately the order of one tenth of a second. Considering a transaction accessing hundred of records, the delay time is substantial.

3. **Communication bottleneck** -

As all requests have to be sent to the central DBMS, the DBMS may create a *bottleneck*. This is another factor that contributes to the high access delay time.

4. **Slow response time and low throughput** -

Slow response time and low transaction throughput are the immediate consequences of the two undesirable features listed above (2 and 3).

5. **Low reliability** -

A single system failure at the central DBMS will effectively halt the operation of the whole system. That could have meant million dollars loss in revenue to an organization (*say an airline*) which maintains thousands of communication terminals.

To resolve the above listed problems and satisfy the users' increasing demand in having fast and easy access of the information, the concept of distributing information across the computer network was conceived. This concept triggered a new area of research, namely **Distributed DBMS.**

One can view a distributed DBMS as a set of centralized DBMSs distributed over a large geographical area, each connected to a same communication network (*See figure 5.2*). The advantages of a distributed DBMS over a centralized DBMS are [DATA83] :-

a. **Reliability** : The operation of a distributed DBMS is not subject to total suspension when an individual site is disabled.

b. **Accessibility** : By storing the data objects at the sites where they are most frequently accessed, a distributed DBMS can reduce both the response times and communication costs; and thus provide a faster and easier access of the data.

c. **Local autonomy** : Each individual site in a distributed DBMS has its own control over the local data and accesses to these data. In the other words, each site is performing as a single centralized DBMS which has communication links with other sites.

d. **Capacity and incremental growth** : Capacity of the data base maintained in a distributed DBMS can be expanded by adding new sites to the communication network. This is easier than updating the processing ability to cope with the expansion of the database in the centralized DBMS. Moreover, the additional new sites also improve the availability of the data in terms of the geographical area.

A distributed DBMS is either *fully replicated*, *partially replicated* or *fully disjoint*. A distributed DBMS is called *fully replicated* if the database maintained

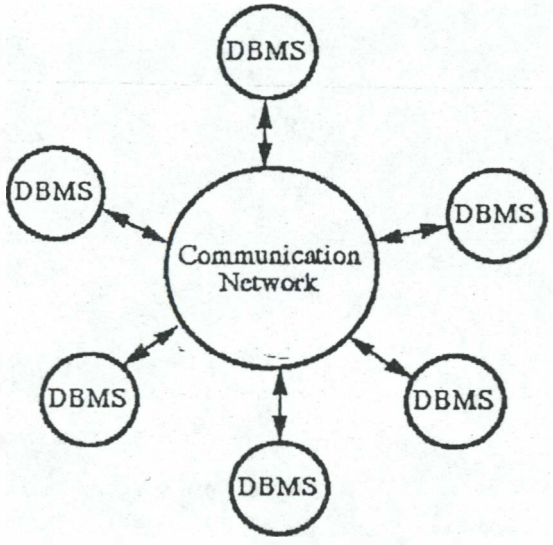


Figure 5.2 Distributed DBMS

in the organization is available at all nodes of the network. With this type of database distribution, read accesses can be carried up efficiently; however, it will also create inefficiency in updating the database, since all copies of the database in the network must be updated to maintain the consistency. Another extreme alternative is to distribute the data objects in such a way that no two local databases maintain a similar data object, that is *fully disjoint*. In this case, read and write requests to access a data object which is not at the local database have to be sent to the site where the data object is stored. This may incur high communication cost and communication overhead. The most preferred alternative is the compromise of these two, which is *partially replicated distribution*. Whether a distributed DBMS is *fully replicated*, *fully disjoint* or *partially replicated*, it can't avoid the concurrency control problems.

Concurrency control problems in the distributed DBMS are much more difficult to deal with than in the centralized environment. In the past ten years, considerable efforts have been contributed to developing algorithms to resolve the

problems. Most of these algorithms are normally complex and without mathematical proofs to support their correctness. In fact, majority of them are the variations of the two basic mechanisms, namely Two-Phase Locking mechanism and Timestamping mechanism[BERN81].

In the rest of this section, problems of applying Two-Phase Locking mechanism and Timestamping mechanism in distributed DBMS are briefly discussed. Descriptions of various distributed concurrency control algorithms can be found in [BERN81].

5.2.1 Two-Phase Locking mechanism

The major problem of applying Two-Phase Locking mechanism in distributed DBMS is to resolve the deadlock. In centralized DBMS, deadlock is either prevented by using preclaim strategy or detected and resolved by using pre-emptive strategy (*see section 3.2.2*). However, these two strategies are inadequate to tackle deadlock in distributed DBMS. The following two examples illustrate the inadequacy of these two strategies.

Example. *Deadlock prevention - Preclaim strategy*

Suppose the following transactions $T1$ and $T2$ are submitted to the concurrency controllers at site $S1$ and $S2$ respectively.

$$T1 : R(x)\, W(y) \;\; and \;\; T2 : R(y)\, W(x)$$

Assume that data object x is available at site $S1$ but not at $S2$, and data object y is stored at site $S2$ but not at $S1$. If both transactions are submitted to their respective concurrency controllers at about the same time, the following situation may occur.

1. $T1$ attempts to lock both data object x and y. Request to lock x is granted immediately but not data object y as y is not stored at site $S1$. Request to lock y is then sent to site $S2$.

2. $T2$ attempts to lock both data object y and x. Request to lock y is granted immediately but not data object x as x is at remote site $S1$. Request to lock x is sent to site $S1$.
3. $T1's$ request to lock y arrives at site $S2$ and finds that y is locked by $T2$ with a conflicting lock. $T1's$ request joins the waiting queue and wait for y to be unlocked.
4. $T2's$ request to lock x arrives at site $S1$ and finds that x is locked by $T1$ with a conflicting lock. $T2's$ request joins the waiting queue and wait for x to be unlocked.
5. **Multi-site Deadlock occurred !!!**

(*Notice the similarity of this example and the Dinning Philosophers problem [DIJK71] which allows a philosopher to hold a fork while waiting for another fork.*)

Due to the communication delay and the property of local autonomy, preclaim strategy cannot guarantee that when a local transaction is attempting to lock all the requested data objects, there is no other remote transaction claiming locks on one or more of these objects. As this condition cannot be fulfilled, preclaim strategy cannot guarantee the prevention of the occurrence of deadlock.

There are several solutions to the above problem. The simplest solution is disallowing the transactions to wait for any remote data object. Abort and restart a transaction once one of its requested remote data objects is found to be locked with a conflicting lock. Of course, this method also introduces unnecessary abortions, indefinite postponement (or repeatedly restart) and a very busy communication traffic.

Another better solution is assigning exactly one copy of each data object maintained in the distributed DBMS as the primary copy of that object. All accesses to a data object must obtain a lock on the primary copy of this object. This effectively provides a kind of central control mechanism similar to centralized DBMS to the system. However, this method will also increase both the response times and communication costs even if a copy of the requested data object is available at the local site (unless it is the primary copy). To improve the reliability

of the DBMS, one can assign another copy of the data object as a secondary copy. This will maintain the availability of the object even when the primary copy is unreachable due to the failure of the site where it is located.

Example. *Deadlock detection - Pre-emptive strategy*

Suppose a similar situation as described in last example arises in a distributed DBMS adopting dynamic Two-Phase Locking mechanism. Similar *multi-site deadlock* may occur without being detected. The following descriptions illustrate the occurrence of the deadlock.

1. $T1's$ request to lock the data object x is granted immediately and the object is read. It attempts to write data object y but finds that y is not stored at $S1$ and thus the request is sent to $S2$.
2. $T2's$ request to lock the data object y is granted immediately and the object is read. It attempts to write data object x but realizes that x is not available at $S2$ and hence the request is sent to $S1$.
3. $T1's$ request to write y arrives at $S2$. Its request to write-lock y is delayed as the lock is conflicting with the read-lock held by $T2$ on y. Therefore, a conflict arc is added to the conflict graph maintained at $S2$. Since the newly added arc does not create a cycle in the graph, there is no deadlock.
4. $T2's$ request to write x arrives at $S1$. Its request to write-lock x is delayed as the lock is conflicting with the read-lock held by $T1$ on x. Therefore, a conflict arc is added to the conflict graph maintained at $S1$. Since the newly added arc does not create a cycle in the graph, there is no deadlock.
5. However, **Multi-site Deadlock has occurred between $T1$ and $T2$!!!**

(*See figure 5.3*

As the conflict graph maintained at each site of the distributed DBMS is only capable of detecting its local deadlocks, multi-site deadlock will not be detected and resolved. One solution to this problem is designating one site as a deadlock detecting site. The responsibility of this site is constructing a **global conflict graph** which maintains the conflicts occurred at all sites of the system. Any occurrence of multi-site deadlock can then be detected by searching for cycle in

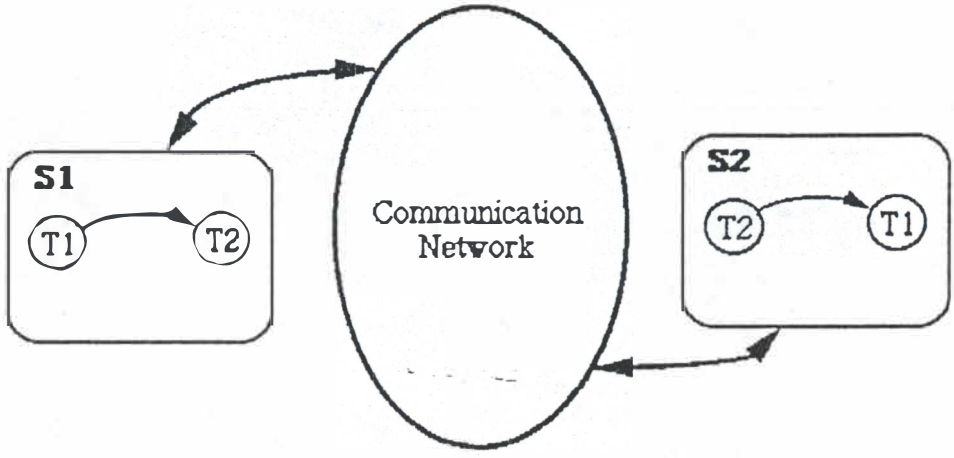


Figure 5.3 Multi-site Deadlock

this global conflict graph. However, any system failure at the deadlock detecting site will immediately disable the systems's ability in detecting multi-site deadlock. Moreover, this method also introduces busy communication traffic as each individual site in the system has to send a copy of its local conflict graph to the deadlock detecting site periodically.

Another solution is called **hierarchical deadlock detection**[16] . This approach is most appropriate for the distributed DBMS organized in a hierarchical network, such as the tree structured network shown in *figure 5.4*. This structure is very common for corporate networks in which local office DBMSs are linked to the area office DBMS, area office DBMSs are linked to the regional office DBMS, and so on. In this network, a parent DBMS and child DBMS can communicate directly, but sibling DBMSs can only communicate with each other through their parent DBMS. Each node in the network maintains a conflict graph. The conflict graph maintained by a leaf node contains all conflicts occur at that node; and the conflict graph maintained by a non-leaf node (or a parent) contains the multi-site conflicts occur among its immediate children. Obviously, in comparison with the

[16] *This is another idea borrowed from distributed Operating Systems.[ABRA88]*

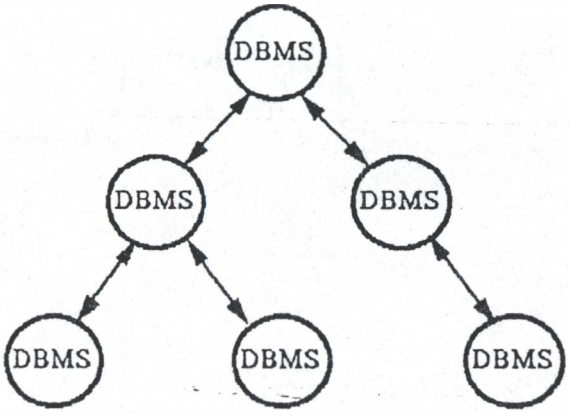


Figure 5.4 Tree Structured Network

previous '*single deadlock detecting site*' method, this method is relatively more reliable and incur lower communication costs.

5.2.2 Timestamping

As discussed in *section 3.3*, the major advantage of Timestamping mechanism is that *deadlock* is impossible to occur and thus deadlock prevention or detection is unnecessary. Due to this feature, Timestamping is relatively easier to be modified to cope with the distributed environment.

Based on Timestamping mechanism, each transaction submitted to the concurrency controller is given a unique timestamp. In distributed DBMS, a timestamp usually consists of a local site identifier as lower order bits and local clock time as higher order bits. One can assign priority to the sites by giving different identifier values. The lower the identifier, the higher the priority of the site.

There are many variations of distributed Timestamping mechanism. Most of them adopts pre-transaction Timestamping mechanism. Basically, pre-transaction distributed Timestamping mechanism has the similar scheduling protocol as its centralized counterpart described in *section 3.3.1*. The only difference is that the timestamp of the submitted transaction has to be broadcasted to all sites at which the data objects accessed by this transaction are stored. Due to the communication

delay, request originated from a younger transaction accessing a local data object may be granted before the concurrency controller receives the timestamp of an older transaction which is accessing the same object from a remote site. Various methods have been proposed to resolve this problem and each of them have their pros and cons. Descriptions and comparisons of these methods can be found in [BERN81][DATE83].

REFERENCES

[ABRA88] **Abraham Silberschatz and J.L Peterson,**
"Operating System Concepts",
Addison-Wesley Publishing Company, 1988.

[ALAG85] **S. Alagic,**
"Relational Database Technology",
Springer-Verlag, 1985.

[ASH75] **E.A. Ashcroft,**
"Proving assertions about parallel programs",
J. Comput. Syst. Sci., Vol 10, 1975.

[BERN80] **P.A. Bernstein,**
"Concurrency Control in a System for Distributed Databases (SDD-1)",
ACM TOD, Vol. 15, No. 1, March 1980.

[BERN81] **P.A. Bernstein and N. Goodman,**
"Concurrency Control in Distributed Database Systems",
Computing Surveys, Vol. 13, No. 2, 1981.

[BRIN73] **P. Brinch Hansen,**
"Operating System Principles",
Englewood Cliffs, N.J. Prentice-Hall, 1973.

[CARR79] **Bernard Carre,**
"Graphs and Networks",
Clarendon Press - Oxford, 1979.

[COUR71] **P.J. Courtois, F. Heymans and D.L. Parnas,**
"Concurrent Control with Readers and Writers",
Comm. ACM, Vol. 14, No. 10, October1971.

[DATE83] **C.J. Date,,**
"An Introduction to Database Systems", Vol. 2,
The Systems Programming Series. 1983.

[DATE86] **C.J. Date,**
"An Introduction to Database Systems", Vol. 1,
The Systems Programming Series. 1986.

[DIJK65] **E.W. Dijkstra,**
"Cooperating Sequential Processes", Programming Languages,
F. Genuys (ed.), Academic Press, NY, 1968.

[DIJK71] **E.W. Dijkstra,**
"Hierarchical Ordering of Sequential Processes",
Acta Informatica, Vol. 1, 1971.

[EICH88a] **M.H. Eich,**
"Graph Directed Locking",
IEEE Trans. on Soft. Eng., Vol. 14, No. 2,
February 1988.

[EICH88b] **M.H. Eich and D.L. Wells,**
"Database Concurrency Control Using Data Flow Graphs",
ACM TODS, Vol. 13, No. 2, June 1988.

[ESWA76] **K.P. Eswaran** *et al.*,
"The Notions of Consistency and Predicate Locks
in a Database System",
Comm. ACM, Nov. 1976, Vol. 19, No. 11.

[FERN81] **E.B. Fernandez, R.C. Summers and C. Wood,**
"Database Security and Integrity",
The Systems Programming Series, 1981.

[FLOYD67] **R.W. Floyd,**
"Assigning meanings to programs",
Proc. Symp. Applied Mathematics,
American Math. Soc., 1967.

[GIFF84] **D. Gifford and A. Spector,**
"The TWA Reservation System - Case Study",
Comm. ACM, Vol. 27, No. 7, July 1984.

[GORD86] **C.E. Gordon,**
"Database Management : Objectives, System Functions,
and Administration",
McGraw-Hill Series in MIS, 1986.

[GRAY78] **J.N. Gray,**
"Notes on Data Base Operating Systems",
Operating Systems : *An Advanced Course*
Lecture Notes in Computer Science, Vol. 60, 1978.

[GRAY81] **J.N. Gray,**
"The Recovery Manager of the System R Data Manager",
ACM Computing Surveys, Vol. 13, No. 2, June 1981.

[HAE83] **T. Haerder and A. Reuter,**
"Principles of Transaction-Oriented Database Recovery",
ACM Computing Surveys, Vol. 15, No. 4, December 1983.

[HOAR74] **C.A.R. Hoare,**
"Monitors : An Operating System Structuring Concept",
Comm. ACM, Vol. 17, No. 10, October 1974.

[KELLER76] **R.M. Keller,**
"Formal Verification of Parallel Programs",
Comm. ACM, Vol. 19, 1976.

[KUNG79] **H.T. Kung and C.H. Papadimitriou,**
"An Optimality Theory of Concurrency Control for Databases",
ACM SIGMOD International Symposium on Management of Data, 1979.

[KUNG81] **H.T. Kung and J.T. Robinson,**
"On Optimistic Concurrency Control",
ACM TODS, Vol. 6, No. 2, 1981.

[LAMP76] **L. Lamport,**
"Towards a Theory of Correctness for Multi-user Data Base Systems",
Technical Report CA-7610-0712,
Massachusetts Computer Associates, Oct. 1976.

[LISTER] **A. lister,**
"Fundamentals of Operating Systems".

[LORIE77] **R.A. Lorie**
"Physical Integrity in a Large Segmented Database",
ACM TODS, Vol. 2, No. 1, March 1977.

[PAPA82] **C.H. Papadimitriou,**
"A Theorem in Database Concurrency Control",
J. ACM, Vol. 29, No. 4, October, 1982.

[PAPA86] **C.H. Papadimitriou,**
"The Theory of Database Concurrency Control",
Computer Science Press, 1986.

[ULLMAN82] **J.D. Ullman,**
"Principles of Database Systems",
Computer Science Press, 1982.

[WOLF85] **R. Wolfgang,**
"Petri Nets - An Introduction",
Spinger-verlag, 1985.

[WOOD87] **J.C.P. Woodcock,**
"Transaction Processing Primitives and CSP",
IBM J. Res. Develop., Vol. 31, No. 5, September 1987.

Appendix A
Dynamic Two-Phase Locking Mechanism

```
program dynamic_2p_lock(input,output,transfile,colours);

type

  string = packed array [1..10] of char;

  colour = ( red, blue, green, orange, yellow, purple );

  lock_flag = (W,R,E);

  lock_ptr = ^lock_list;

  lock_list = record
                Tid : integer;
                Tlock : lock_flag;
                next : lock_ptr;
              end;

  lockbit = record
              s : integer;
              wait_que : lock_ptr;
              w_tail : lock_ptr;
              granted_que : lock_ptr;
              g_tail : lock_ptr;
            end;

  DGptr = ^DG;

  conflict_ptr = ^conflict_list;

  object_ptr = ^object;

  object = record
             access_key : string;
             next : object_ptr;
           end;

  DG = record
         id : integer;
         wait_trans : integer;
         object_list : object_ptr;
         last_object : object_ptr;
         conflict_head : conflict_ptr;
         last_conflict : conflict_ptr;
         next : DGptr;
       end;
```

```
  conflict_list = record
                    access_key : string;
                    transaction : DGptr;
                    next : conflict_ptr;
                  end;

  transid_ptr = ^Tlist;

  Tlist = record
            Tid : integer;
            next : transid_ptr;
          end;

  colour_record = record
    name :          [KEY(0)] string;
    favourite  :    colour;
    lock_bit : lockbit;
    end {colour_record};

  trans_step = record
    id      : integer;
    case access : lock_flag of
                  R,W : (access_key : string);
                  E : ();
    end;

var
  ch : char;
  colours:    file of colour_record;
  request : trans_step;
  conflict_graph,last_node: DGptr;
  transfile : text;

PROCEDURE initialization;
begin
  OPEN(colours,'colours.dat',unknown,
       access_method := keyed,
       organization := indexed       );

  reset(transfile);
  conflict_graph := NIL;
  last_node := conflict_graph;
end;

PROCEDURE get_next_request(Var data_step : trans_step);
begin
  with data_step do
    begin
      read(transfile,id,access);
      case access of
        R,W : readln(transfile,ch,access_key);
        E : readln(transfile);
```

```
      end;
    end;
end;

PROCEDURE update_conflict_graph (request : trans_step);
var Ti_node : DGptr;
    temp_obj : object_ptr;

Function create_new_node (request : trans_step) : DGptr;
  var temp : DGptr;
begin
  NEW(temp);
  with temp^ do begin
    id := request.id;
    wait_trans := 0;
    object_list := temp_obj;
    last_object := object_list;
    conflict_head := NIL;
    last_conflict := conflict_head;
    next := NIL;
  end;
  create_new_node := temp;
end;

Function node_found (i: integer; Var i_node : DGptr) : Boolean;
  Var found : boolean;
      temp : DGptr;
begin
  found := false;
  temp := conflict_graph;
  while (temp <> NIL) and (not found) do
    begin
      found := (temp^.id = request.id);
      if not found then temp := temp^.next;
    end;
  if found then i_node := temp;
  node_found := found;
end;

begin

  if conflict_graph = NIL then
    begin
      Ti_node := create_new_node(request);
      conflict_graph := Ti_node;
      last_node := Ti_node;
    end
  else
    if not node_found(request.id,Ti_node) then
      begin
        Ti_node := create_new_node(request);
        last_node^.next := Ti_node;
        last_node := last_node^.next;
```

```
      end
    else
      with Ti_node^ do begin
        NEW(temp_obj);
        temp_obj^.access_key := request.access_key;
        temp_obj^.next := NIL;
        last_object^.next := temp_obj;
        last_object := last_object^.next;
      end;
end;

FUNCTION compatible(     Tid      : integer;
                         Tlock    : lock_flag;
                         w_que    : lock_ptr;
                         g_que    : lock_ptr;
                  var conf_list : transid_ptr ) : boolean;
var last : transid_ptr;
    is_compatible : boolean;

  PROCEDURE add_conflict_list (confid : integer);
    VAR temp : transid_ptr;
  begin
    NEW(temp);
    temp^.Tid := confid;
    temp^.next := NIL;
    if conf_list = NIL then
      begin
        conf_list := temp;
        last := conf_list;
      end
    else
      begin
        last^.next := temp;
        last := last^.next;
      end;
  end;

begin
  if (Tlock = W) then
    begin
      is_compatible := (g_que = NIL);
      while (g_que <> NIL) do begin
        if Tid <> g_que^.Tid then
          add_conflict_list(g_que^.Tid)
        else is_compatible := True;
        g_que := g_que^.next;
      end;
      is_compatible := is_compatible and (conf_list = NIL);
      if not is_compatible then begin
        while (w_que <> NIL) and (w_que^.Tid <> Tid) do begin
          add_conflict_list(w_que^.Tid);
          w_que := w_que^.next;
```

```
        end;
      end;
    end
  else
    begin
      is_compatible := (g_que = NIL);
      while (g_que <> NIL) do begin
         if (Tid <> g_que^.Tid) and (g_que^.Tlock = W) then
           add_conflict_list(g_que^.Tid)
         else if (Tid = g_que^.Tid) then is_compatible := True;
         g_que := g_que^.next;
      end;
      while (w_que <> NIL) and (w_que^.Tid <> Tid) do begin
         if (w_que^.Tlock = W)
         then add_conflict_list(w_que^.Tid);
         w_que := w_que^.next;
      end;
      if not is_compatible
      then is_compatible := (conf_list = NIL);
    end;
  compatible := is_compatible;
end;

PROCEDURE end_transaction (Tid : integer);
  var pre_DG,temp_DG : DGptr;

  PROCEDURE check_wait_request_state;
    var this,last,temp : lock_ptr;
        clash_list : transid_ptr;
  begin
    with colours^ do begin
      this := lock_bit.wait_que;
      last := this;
      clash_list := NIL;

      while this <> NIL do
        if compatible(this^.Tid,this^.Tlock,lock_bit.wait_que,
                          lock_bit.granted_que,clash_list) then
          begin

            { send to data manager }
            with this^ do writeln(Tid,Tlock,name);
            { join granted queue }
            NEW(temp);
            temp^.Tid := this^.Tid;
            temp^.Tlock := this^.Tlock;
            temp^.next := NIL;
            if lock_bit.granted_que = NIL then begin
              lock_bit.granted_que := temp;
              lock_bit.granted_que^.next := NIL;
              lock_bit.g_tail := lock_bit.granted_que;
            end
```

```
            else
            begin
              lock_bit.g_tail^.next := temp;
              lock_bit.g_tail := lock_bit.g_tail^.next;
              lock_bit.g_tail^.next := NIL;
            end;

            { remove request from wait queue }
            if last = this then begin
              lock_bit.wait_que := lock_bit.wait_que^.next;
              this := lock_bit.wait_que;
              last := this;
            end
            else
            begin
              last^.next := this^.next;
              this := last^.next;
            end;
          end
        else
          begin
            last := this;
            this := this^.next;
          end;

      if lock_bit.wait_que = NIL then
         lock_bit.w_tail := lock_bit.wait_que;
      if lock_bit.granted_que = NIL then
         lock_bit.g_tail := lock_bit.granted_que;
    end;
  end;

  PROCEDURE db_unlock;
    var this,last : lock_ptr;
  begin
    with colours^ do begin
      this := lock_bit.granted_que;
      last := this;

      while this <> NIL do begin
        if this^.Tid = Tid then begin
          lock_bit.s := lock_bit.s + 1;

          { remove request }
          if this <> last then begin
            last^.next := this^.next;
            if this = lock_bit.g_tail
            then lock_bit.g_tail := last;
            dispose(this);
            this := last^.next;
          end
          else begin
            lock_bit.granted_que := lock_bit.granted_que^.next;
            if this = lock_bit.g_tail
```

```
            then lock_bit.g_tail := NIL;
            dispose(this);
            this := lock_bit.granted_que;
            last := this;
          end;
        end;
        if this <> NIL then this := this^.next;
      end;

      check_wait_request_state;
    end;
  end;

begin
    { remove node }
    temp_DG := conflict_graph;
    if temp_DG^.id = Tid
    then conflict_graph := conflict_graph^.next
    else
    begin
      pre_DG := temp_DG;
      temp_DG := temp_DG^.next;
      WHILE temp_DG^.id <> Tid do
        begin
          pre_DG := temp_DG;
          temp_DG := temp_DG^.next;
        end;
      pre_DG^.next := temp_DG^.next;
      if temp_DG = last_node then last_node := pre_DG;
    end;
    if conflict_graph = NIL then last_node := conflict_graph;

    with temp_DG^ do begin
      while conflict_head <> NIL do begin
        conflict_head^.transaction^.wait_trans :=
          conflict_head^.transaction^.wait_trans - 1;
        conflict_head := conflict_head^.next;
      end;
      while object_list <>NIL do begin
        findk(colours,0,object_list^.access_key);
        if ufb(colours) then
          writeln('Sorry - that name is not in the file');
        db_unlock;
        update(colours);
        object_list := object_list^.next;
      end;
    end;
    dispose(temp_DG);
end;

PROCEDURE add_arc_conflict_graph (Tid        : integer;
                                  conf_list  : transid_ptr;
                                  clash_key  : string);
```

```
  var node,Ti_node : DGptr;
      conflict_arc : conflict_ptr;
      previous,this : lock_ptr;
      deadlock : boolean;

  FUNCTION path (Tid : integer;
                 trans_conf : conflict_ptr) : boolean;
    var is_path : boolean;
        previous : integer;
  begin
    if Tid = trans_conf^.transaction^.id then
     begin
       writeln('cycle -> ',Tid);
       path := true
     end
    else
    begin
      is_path := false;
      previous := trans_conf^.transaction^.id;
      trans_conf := trans_conf^.transaction^.conflict_head;

      while (trans_conf <> NIL) and (not is_path) do
        begin
          is_path := path(Tid,trans_conf);
          if not is_path then
            begin
              previous := trans_conf^.transaction^.id;
              trans_conf := trans_conf^.next;
            end
          else writeln('cycle -> ',previous);
        end;
      path := is_path;
    end;
  end;

begin
  Ti_node := conflict_graph;
  while (Ti_node^.id <> Tid) do
    Ti_node := Ti_node^.next;
  deadlock := false;

  while conf_list <> NIL do begin
    node := conflict_graph;
    while node <> NIL do begin
      if node^.id = conf_list^.Tid then begin
        Ti_node^.wait_trans := Ti_node^.wait_trans + 1;
        { create new conflict arc }
        NEW(conflict_arc);
        conflict_arc^.access_key := clash_key;
        conflict_arc^.transaction := Ti_node;
        conflict_arc^.next := NIL;
        if node^.conflict_head = NIL
        then node^.conflict_head := conflict_arc;
```

```
          node^.last_conflict := conflict_arc;
          node^.last_conflict := node^.last_conflict^.next;

          if (Ti_node^.conflict_head <> NIL) then
          if path(conf_list^.Tid,Ti_node^.conflict_head) then
            begin
              writeln('cycle -> ',Tid,conf_list^.Tid);
              writeln('deadlock detected !!!');
              writeln('abort transaction ',Tid:1);

              { remove arc }
              Ti_node^.wait_trans := 0;
              with colours^ do begin
                this := lock_bit.wait_que;
                previous := this;
                while this <> lock_bit.w_tail do begin
                  previous := this;
                  this := this^.next;
                end;
                if previous = this then begin
                  lock_bit.wait_que := NIL;
                  previous := NIL;
                end
                else previous^.next := NIL;
                lock_bit.w_tail := previous;
                dispose(this);
                lock_bit.s := lock_bit.s + 1;
              end;

              end_transaction(Tid);

              node^.last_conflict := NIL;
              dispose(conflict_arc);
              deadlock := true;
            end;
        end;
        node := node^.next;
      end;
      conf_list := conf_list^.next;
    end;
    if not deadlock then update(colours);
  end;

  PROCEDURE dblock ( R : trans_step);
    var temp,this : lock_ptr;
        conflict_list : transid_ptr;

    FUNCTION no_other_granted_request
                (Tid : integer; Tlock : lock_flag) : BOOLEAN;
      var temp : lock_ptr;
    begin
      with colours^ do begin
        temp := lock_bit.granted_que;
```

```
      while (temp <> NIL) and (temp^.Tid <> Tid) do
        temp := temp^.next;
      if temp <> NIL then begin
        if Tlock = W then temp^.Tlock := W;
        no_other_granted_request := false;
      end
      else no_other_granted_request := true;
    end;
  end;

  PROCEDURE send_to_data_manager ;
  begin
    with R do writeln(id,access,access_key);
  end;

begin
  findk(colours,0,R.access_key);
  if ufb(colours) then
  writeln('Sorry - that name is not in the file');

  with colours^ do begin
    NEW(temp);
    temp^.Tid := R.id;
    temp^.Tlock := R.access;
    temp^.next := NIL;

    if lock_bit.s > 0 then
    begin
      send_to_data_manager;

      { join granted queue }
      lock_bit.granted_que := temp;
      lock_bit.g_tail := lock_bit.granted_que;
      lock_bit.s := lock_bit.s - 1;
      update(colours);
    end
    else
    begin
      conflict_list := NIL;
      if compatible(R.id,R.access,lock_bit.wait_que,
                    lock_bit.granted_que,conflict_list)
      then
      begin
        send_to_data_manager;
        if no_other_granted_request(R.id,temp^.Tlock) then
          begin
            { join granted queue }
            lock_bit.g_tail^.next := temp;
            lock_bit.g_tail := lock_bit.g_tail^.next;

            lock_bit.s := lock_bit.s - 1;
          end;
        update(colours);
```

```
        end
        else
        begin
         { join wait queue }
         if lock_bit.wait_que = NIL then begin
            lock_bit.wait_que := temp;
            lock_bit.w_tail := lock_bit.wait_que;
          end
          else
          begin
            lock_bit.w_tail^.next := temp;
            lock_bit.w_tail := lock_bit.w_tail^.next;
          end;

          lock_bit.s := lock_bit.s - 1;
          add_arc_conflict_graph(R.id,conflict_list,name);
        end;
      end;
    end;
  end;

begin
  initialization;
  repeat
       get_next_request(request);
       case request.access of
  W,R: begin
                 update_conflict_graph(request);
                 dblock(request);
               end;
  E: end_transaction(request.id);
       end {case};
  until eof(transfile);
end.
```

Appendix B
Pre-Transaction Two-Phase Locking Mechanism

```
program pre_transaction_2p_lock
        (input,output,transfile,colours);

type

  string = packed array [1..10] of char;

  colour = ( red, blue, green, orange, yellow, purple );

  access_flag = (R,W,L,E);

  lock_ptr = ^lock_list;

  lock_list = record
                Tid : integer;
                Tlock : access_flag;
                next : lock_ptr;
              end;

  lockbit = record
              s : integer;
              wait_que : lock_ptr;
              w_tail : lock_ptr;
              granted_que : lock_ptr;
              g_tail : lock_ptr;
            end;

  object_ptr = ^object;

  object = record
             access_key : string;
             next : object_ptr;
           end;

  trans_step = record
    id     : integer;
    case access : access_flag of
                  R,W : (access_key : string);
                  L,E : ();
    end;

  active_ptr = ^active_trans;

  active_trans = record
                   id : integer;
```

```
                    wait_request_no : integer;
                    object_list : object_ptr;
                    last_object : object_ptr;
                    next : active_ptr;
                  end;

  transid_ptr = ^Tlist;

  Tlist = record
            Tid : integer;
            next : transid_ptr;
          end;

  colour_record = record
    name :          [KEY(0)] string;
    favourite  :    colour;
    lock_bit : lockbit;
    end {colour_record};

var
  ch : char;
  colours:    file of colour_record;
  request : trans_step;
  transaction_list,last_transaction : active_ptr;
  transfile : text;

PROCEDURE initialization;
begin
  OPEN(colours,'colours.dat',unknown,
       access_method := keyed,
       organization := indexed       );

  reset(transfile);
  transaction_list := NIL;
  last_transaction := transaction_list;
end;

PROCEDURE get_next_request (Var request : trans_step);
begin
     with request do
        begin
          read(transfile,id,access);
          case access of
            R,W : readln(transfile,ch,access_key);
            L   : ;
            E   : readln(transfile);
          end;
        end;
end;

PROCEDURE signal_all_lock_granted (Tid : integer);
begin
  writeln('All locks requested by transaction '
```

```
                                    ,Tid:1,' granted.');
  writeln('Signal to send rest of transaction !!!');
  writeln;
end;

PROCEDURE find_object (object_access_key : string);
begin
  findk(colours,0,object_access_key);
  if ufb(colours) then
    writeln('Sorry - that name is not in the file');
end;

FUNCTION compatible(     Tid      : integer;
                         Tlock    : access_flag;
                         w_que    : lock_ptr;
                         g_que    : lock_ptr;
                  var conf_list : transid_ptr ) : boolean;
var last : transid_ptr;

  PROCEDURE add_conflict_list (confid : integer);
    VAR temp : transid_ptr;
  begin
    NEW(temp);
    temp^.Tid := confid;
    temp^.next := NIL;
    if conf_list = NIL then
      begin
        conf_list := temp;
        last := conf_list;
      end
    else
      begin
        last^.next := temp;
        last := last^.next;
      end;
  end;

begin
  if (Tlock = W) then
    begin
      while (g_que <> NIL) do begin
        if Tid <> g_que^.Tid then
          add_conflict_list(g_que^.Tid);
        g_que := g_que^.next;
      end;
      while (w_que <> NIL) and (w_que^.Tid <> Tid) do begin
        add_conflict_list(w_que^.Tid);
        w_que := w_que^.next;
      end;
    end
  else
    begin
```

```
      while (g_que <> NIL) do begin
         if (Tid <> g_que^.Tid) and (g_que^.Tlock = W) then
           add_conflict_list(g_que^.Tid);
         g_que := g_que^.next;
      end;
      while (w_que <> NIL) and (w_que^.Tid <> Tid) do begin
         if (w_que^.Tlock = W)
         then add_conflict_list(w_que^.Tid);
         w_que := w_que^.next;
      end;
    end;
  compatible := (conf_list = NIL);
end;

PROCEDURE dblock (     Tid          : integer;
                       access       : access_flag;
                  Var wait_count    : integer);
  var temp : lock_ptr;
      conflict_list : transid_ptr;

  FUNCTION in_granted_queue
                    ( Tid : integer;
                      Tlock : access_flag) : BOOLEAN;
    var temp : lock_ptr;
  begin
    with colours^ do begin
      temp := lock_bit.granted_que;
      while (temp <> NIL) and (temp^.Tid <> Tid) do
        temp := temp^.next;
      if temp <> NIL then begin
        if Tlock = W then temp^.Tlock := W;
        in_granted_queue := true;
      end
      else in_granted_queue := false;
    end;
  end;

begin
  with colours^ do begin
    NEW(temp);
    temp^.Tid := Tid;
    temp^.Tlock := access;
    temp^.next := NIL;

    if lock_bit.s > 0 then
    begin
      { join granted queue }
      lock_bit.granted_que := temp;
      lock_bit.g_tail := lock_bit.granted_que;
      lock_bit.s := lock_bit.s - 1;
      update(colours);
    end
```

```
    else
    begin
      conflict_list := NIL;
      if compatible(Tid,access,lock_bit.wait_que,
                    lock_bit.granted_que,conflict_list)
      then
      begin
        if not in_granted_queue(Tid,temp^.Tlock) then
          begin
            { join granted queue }
            lock_bit.g_tail^.next := temp;
            lock_bit.g_tail := lock_bit.g_tail^.next;
            lock_bit.s := lock_bit.s - 1;
          end;
        update(colours);
      end
      else
      { not compatible }
      begin
        { join waiting queue }
        if lock_bit.wait_que = NIL then begin
          lock_bit.wait_que := temp;
          lock_bit.w_tail := lock_bit.wait_que;
        end
        else
        begin
          lock_bit.w_tail^.next := temp;
          lock_bit.w_tail := lock_bit.w_tail^.next;
        end;
        lock_bit.s := lock_bit.s - 1;
        wait_count := wait_count + 1;
        update(colours);
      end;
    end;
  end;
end;

PROCEDURE new_transaction (Tid : integer);
var node : active_ptr;
    access_type : access_flag;

  PROCEDURE lock_access_object
            ( Tid : integer;
              Var object_list,last_object : object_ptr;
              Var wait_count  : integer);
    var access_type : access_flag;
        object_access_key : string ;
        cnt,count : integer;
        temp_object : object_ptr;
  begin
    while not eoln(transfile) do begin
      read(transfile,access_type,count,ch);
```

```
      for cnt := 1 to count do begin
        read(transfile,object_access_key);
        NEW(temp_object);
        with temp_object^ do begin
          temp_object^.access_key := object_access_key;
          temp_object^.next := NIL;
        end;

        { add object to obejct list }
        if object_list = NIL then
          begin
            object_list := temp_object;
            last_object := object_list;
          end
        else
          begin
            last_object^.next := temp_object;
            last_object := last_object^.next;
          end;
        find_object(object_access_key);
        dblock(Tid,access_type,wait_count);
      end;
    end;
    readln(transfile);
    if wait_count = 0 then signal_all_lock_granted(Tid);
  end;

begin

  NEW(node);
  with node^ do begin
    id := Tid;
    wait_request_no := 0;
    object_list := NIL;
    last_object := NIL;
    next := NIL;

    { attempts to lock all data objects accessed }
    lock_access_object(Tid,object_list,last_object,
                       wait_request_no);
  end;

  { add new transaction node to the active transaction list}
  if transaction_list = NIL then
    begin
      transaction_list := node;
      last_transaction := transaction_list;
    end
  else
    begin
      last_transaction^.next := node;
      last_transaction := last_transaction^.next;
    end
```

```
end;

PROCEDURE send_to_data_manager (step : trans_step);
begin
  with step do writeln(id,access,access_key)
end;

PROCEDURE end_transaction (Tid : integer);

  var previous,this : active_ptr;

  PROCEDURE check_wait_request_state;
    var this,last,temp : lock_ptr;
        clash_list : transid_ptr;
        trans_node : active_ptr;
  begin
    with colours^ do begin
      this := lock_bit.wait_que;
      last := this;
      clash_list := NIL;

      while this <> NIL do
        if compatible(this^.Tid,this^.Tlock,lock_bit.wait_que,
                      lock_bit.granted_que,clash_list) then
          begin
            { join granted queue }
            NEW(temp);
            temp^.Tid := this^.Tid;
            temp^.Tlock := this^.Tlock;
            temp^.next := NIL;
            if lock_bit.granted_que = NIL then begin
              lock_bit.granted_que := temp;
              lock_bit.granted_que^.next := NIL;
              lock_bit.g_tail := lock_bit.granted_que;
            end
            else
            begin
              lock_bit.g_tail^.next := temp;
              lock_bit.g_tail := lock_bit.g_tail^.next;
              lock_bit.g_tail^.next := NIL;
            end;

            trans_node := transaction_list;
            while trans_node^.id <> this^.Tid do
              trans_node := trans_node^.next;
            trans_node^.wait_request_no :=
                trans_node^.wait_request_no - 1;
            if trans_node^.wait_request_no = 0 then
              signal_all_lock_granted(trans_node^.id);

            { remove request from waiting queue }
            if last = this then begin
              lock_bit.wait_que := lock_bit.wait_que^.next;
```

```
            this := lock_bit.wait_que;
            last := this;
          end
          else
          begin
            last^.next := this^.next;
            this := last^.next;
          end;
        end
      else
      {not compatible}
        begin
          last := this;
          this := this^.next;
        end;

    if lock_bit.wait_que = NIL then
       lock_bit.w_tail := lock_bit.wait_que;
    if lock_bit.granted_que = NIL then
       lock_bit.g_tail := lock_bit.granted_que;
  end;
end;

PROCEDURE db_unlock;
  var this,last : lock_ptr;
begin
  with colours^ do begin
    this := lock_bit.granted_que;
    last := this;

    while this <> NIL do begin
      if this^.Tid = Tid then begin
        lock_bit.s := lock_bit.s + 1;

        { remove request }
        if this <> last then begin
          last^.next := this^.next;
          if this = lock_bit.g_tail
          then lock_bit.g_tail := last;
          dispose(this);
          this := last^.next;
        end
        else begin
          lock_bit.granted_que := lock_bit.granted_que^.next;
          if this = lock_bit.g_tail
          then lock_bit.g_tail := NIL;
          dispose(this);
          this := lock_bit.granted_que;
          last := this;
        end;
      end;
      if this <> NIL then this := this^.next;
    end;
```

```
      check_wait_request_state;
    end;
  end;

begin
  { remove node from transaction list }
  this := transaction_list;
  previous := this;
  while this^.id <> Tid do begin
    previous := this;
    this := this^.next;
  end;
  if previous = this then
    transaction_list := transaction_list^.next
  else
    previous^.next := this^.next;
  if last_transaction = this then
    last_transaction := this^.next;

  with this^ do
    while object_list <>NIL do begin
      find_object(object_list^.access_key);
      db_unlock;
      update(colours);
      object_list := object_list^.next;
    end;
  dispose(this);
end;

begin
  initialization;
  repeat
       get_next_request(request);
       case request.access of
          L : new_transaction(request.id);
  W,R: send_to_data_manager(request);
  E: end_transaction(request.id);
       end {case};
  until eof(transfile);
end.
```

Appendix C
Dynamic Timestamping Mechanism

```
program dynamic_timestamp(input,output,timefile,colours);

type

  string = packed array [1..10] of char;

  colour = ( red, blue, green, orange, yellow, purple );

  access_flag = (R,W);

  timebit = record
              read_timestamp,
              write_timestamp : integer;
            end;

  trans_ptr = ^Tlist;

  Tlist = record
            Tid : integer;
            timestamp : integer;
            next : trans_ptr;
          end;

  colour_record = record
                    name :    [KEY(0)] string;
                    favourite  :   colour;
                    time_bit : timebit;
                  end {colour_record};

  trans_request = record
                    id     : integer;
                    access : access_flag;
                    access_key : string;
                  end;

var
  ch : char;
  colours :    file of colour_record;
  timestamp : integer;
  transaction_list,last_transaction : trans_ptr;
  request : trans_request;
  timefile : text;

PROCEDURE initialization;
begin
  OPEN(colours,'colours.dat',unknown,
```

```
        access_method := keyed,
        organization := indexed        );

  reset(timefile);
end;

PROCEDURE get_next_request (var request : trans_request);
begin
   with request do
        readln(timefile,id,access,ch,access_key);
end;

PROCEDURE send_to_data_manager (request : trans_request);
begin
  with request do writeln(id,access,access_key)
end;

PROCEDURE abort_transaction ( Tid : integer );
begin
  writeln('Abort transaction ',Tid:1,' !!!');
end;

PROCEDURE find_object (object_access_key : string);
begin
  findk(colours,0,object_access_key);
  if ufb(colours) then
    writeln('Sorry - that name is not in the file');
end;

FUNCTION transaction_timestamp (Tid : integer) : integer;
  var temp : trans_ptr;
      found : boolean;
begin
  if transaction_list = NIL then
    begin
      NEW(temp);
      with temp^ do begin
        Tid := request.id;
        timestamp := 1;
        next := NIL;
      end;
      transaction_list := temp;
      last_transaction := transaction_list;
      transaction_timestamp := 1;
    end
  else
    begin
      found := false;
      temp := transaction_list;
      while (temp <> NIL) and (not found) do
        begin
          found := (temp^.Tid = request.id);
```

```
          if not found then temp := temp^.next;
        end;
      if not found then
        begin
          NEW(temp);
          with temp^ do begin
            Tid := request.id;
            timestamp := last_transaction^.timestamp + 1;
            next := NIL;
            transaction_timestamp := timestamp;
          end;
          last_transaction^.next := temp;
          last_transaction := last_transaction^.next;
        end
      else
        transaction_timestamp := temp^.timestamp;
    end;
end;

PROCEDURE process_request
          ( request : trans_request;
            timestamp : integer);

  PROCEDURE read_protocol
            (request : trans_request;
             timestamp : integer);
  begin
    with colours^ do begin
      if timestamp >= time_bit.write_timestamp then
        begin
          send_to_data_manager(request);
          time_bit.read_timestamp := timestamp;
          update(colours);
        end
      else
        abort_transaction(request.id);
    end;
  end;

  PROCEDURE write_protocol
            (request : trans_request;
             timestamp : integer);
  begin
    with colours^ do begin
      if (timestamp >= time_bit.write_timestamp) and
            (timestamp >= time_bit.read_timestamp) then
        begin
          send_to_data_manager(request);
          time_bit.write_timestamp := timestamp;
          update(colours);
        end
```

```
      else
        abort_transaction(request.id);
    end;
  end;

begin
  find_object(request.access_key);
  case request.access of
    R : read_protocol(request,timestamp);
    W : write_protocol(request,timestamp);
  end;
end;

begin
  initialization;
  repeat
    get_next_request(request);
    timestamp := transaction_timestamp(request.id);
    process_request(request,timestamp);
  until eof(timefile);
end.
```

Appendix D
Pre-Transaction Timestamping Mechanism

```
program timestamp(input,output,timefile,colours);

type

  string = packed array [1..10] of char;

  colour = ( red, blue, green, orange, yellow, purple );

  access_flag = (R,W,P);

  trans_request = record
                    id     : integer;
                    access : access_flag;
                    access_key : string;
                  end;

  timestamp_ptr = ^time_stamp;

  time_stamp = record
                 stamp : integer;
                 Tid : integer;
                 arrived : boolean;
                 { indicates if the request has arrived }
                 next : timestamp_ptr;
               end;

  timebit = record
              read_timestamp_que,
              last_read_stamp,
              write_timestamp_que,
              last_write_stamp : timestamp_ptr;
            end;

  trans_ptr = ^Tlist;

  Tlist = record
            Tid : integer;
            timestamp : integer;
            next : trans_ptr;
          end;

  colour_record = record
                    name :    [KEY(0)] string;
                    favourite  :   colour;
                    time_bit : timebit;
                  end {colour_record};
```

```
var
  ch : char;
  colours :    file of colour_record;
  timestamp : integer;
  transaction_list,last_transaction : trans_ptr;
  request : trans_request;
  timefile : text;

PROCEDURE initialization;
begin
  OPEN(colours,'colours.dat',unknown,
       access_method := keyed,
       organization := indexed       );

  reset(timefile);
end;

PROCEDURE get_next_request (var request : trans_request);
begin
   with request do begin
     read(timefile,id,access);
     if access in [R,W] then readln(timefile,ch,access_key);
   end;
end;

PROCEDURE send_to_data_manager (request : trans_request);
begin
  with request do writeln(id,access,access_key)
end;

PROCEDURE find_object (object_access_key : string);
begin
  findk(colours,0,object_access_key);
  if ufb(colours) then
    writeln('Sorry - that name is not in the file');
end;

FUNCTION transaction_timestamp (Tid : integer) : integer;
  var node : trans_ptr;
      found : boolean;
begin
  if transaction_list = NIL then
    { empty transaction list }
    begin
      { create new transaction node }
      NEW(node);
      with node^ do begin
        Tid := request.id;
        timestamp := 1;  { the oldest transaction }
        next := NIL;
      end;
```

```
      { join transaction list }
      transaction_list := node;
      last_transaction := transaction_list;

      transaction_timestamp := 1;
    end
  else  { transaction list not empty }
    begin
      { find node }
      found := false;
      node := transaction_list;
      while (node <> NIL) and (not found) do
        begin
          found := (node^.Tid = request.id);
          if not found then node := node^.next;
        end;

      if not found then
        { node not found }
        begin
          { create new transaction node }
          NEW(node);
          with node^ do begin
            Tid := request.id;
            timestamp := last_transaction^.timestamp + 1;
            next := NIL;
            transaction_timestamp := timestamp;
          end;
          last_transaction^.next := node;
          last_transaction := last_transaction^.next;
        end
      else { transaction node found }
        transaction_timestamp := node^.timestamp;
    end;
end;

PROCEDURE claim_all_object (Tid,timestamp : integer);
  var access_type : access_flag;
      object_access_key : string ;
      cnt,count : integer;

  PROCEDURE join_timestamp_queue(id,timestamp : integer;
                                 access_type : access_flag);
    var timestamp_queue,
        last_timestamp,
        temp : timestamp_ptr;
  begin
    with colours^ do begin
      if access_type = R then begin
        timestamp_queue := time_bit.read_timestamp_que;
        last_timestamp := time_bit.last_read_stamp;
      end
```

```
        else
        begin
          timestamp_queue := time_bit.write_timestamp_que;
          last_timestamp := time_bit.last_write_stamp;
        end;

        NEW(temp);
        with temp^ do begin
          stamp := timestamp;
          Tid := id;
          next := NIL;
        end;

        if timestamp_queue = NIL then begin
          timestamp_queue := temp;
          last_timestamp := timestamp_queue;
        end
        else
        begin
          last_timestamp^.next := temp;
          last_timestamp := last_timestamp^.next;
        end;

        if access_type = R then begin
          time_bit.read_timestamp_que := timestamp_queue;
          time_bit.last_read_stamp := last_timestamp;
        end
        else
        begin
          time_bit.write_timestamp_que := timestamp_queue;
          time_bit.last_write_stamp := last_timestamp;
        end;
      end;

      update(colours);
    end;

begin
  while not eoln(timefile) do begin
    read(timefile,access_type,count,ch);
    for cnt := 1 to count do begin
      read(timefile,object_access_key);

      find_object(object_access_key);
      join_timestamp_queue(Tid,timestamp,access_type);
    end;
  end;
  readln(timefile);
end;

PROCEDURE process_request
          ( request : trans_request;
            timestamp : integer);
```

```
PROCEDURE remove_time_stamp(timestamp : integer;
                            var time_bit : timebit;
                            access : access_flag );
  var previous,this,last_stamp,temp : timestamp_ptr;
begin
  if access = R then begin
    temp := time_bit.read_timestamp_que;
    last_stamp := time_bit.last_read_stamp;
  end
  else
  begin
    temp := time_bit.write_timestamp_que;
    last_stamp := time_bit.last_write_stamp;
  end;

  this := temp;
  previous := this;
  while this^.stamp <> timestamp do begin
    previous := this;
    this := this^.next;
  end;

  if this <> previous then begin
    previous^.next := this^.next;
    if this = last_stamp then last_stamp := previous;
    dispose(this);
  end
  else
  begin
    temp := temp^.next;
    if this = last_stamp then last_stamp := NIL;
    dispose(this);
  end;

  if access = R then begin
    time_bit.read_timestamp_que := temp;
    time_bit.last_read_stamp := last_stamp;
  end
  else
  begin
    time_bit.write_timestamp_que := temp;
    time_bit.last_write_stamp := last_stamp;
  end;
end;

PROCEDURE update_time_stamp(timestamp : integer;
                            var time_bit : timebit;
                            access : access_flag);
  var temp : timestamp_ptr;
begin
  if access = R then temp := time_bit.read_timestamp_que
  else temp := time_bit.write_timestamp_que;
```

```
  while temp^.stamp <> timestamp do
    temp := temp^.next;

  temp^.arrived := TRUE;
end;

PROCEDURE read_protocol
          (request:trans_request;
           timestamp:integer);forward;

PROCEDURE write_protocol
          (request:trans_request;
           timestamp:integer);forward;

PROCEDURE check_wait_queue (access : access_flag);
 var timestamp_node : timestamp_ptr;
     request : trans_request;
begin
  with colours^ do begin
    if access = W then begin
      timestamp_node := time_bit.read_timestamp_que;
      while timestamp_node <> NIL do begin
        if timestamp_node^.arrived then begin
          { re-create the request }
          request.id := timestamp_node^.Tid;
          request.access := R;
          request.access_key := name;

          { re-submit the request to read_protocol }
          read_protocol(request,timestamp_node^.stamp);
        end;
        timestamp_node := timestamp_node^.next;
      end;
    end;

    timestamp_node := time_bit.write_timestamp_que;
    while timestamp_node <> NIL do begin
      if timestamp_node^.arrived then begin
        { re-create the request }
        request.id := timestamp_node^.Tid;
        request.access := W;
        request.access_key := name;

        { re-submit the request to the write_protocol }
        write_protocol(request,timestamp_node^.stamp);
      end;
      timestamp_node := timestamp_node^.next;
    end
  end
end;

PROCEDURE read_protocol ;
```

```
    { (request : trans_request; timestamp : integer) }

  begin
    with colours^ do begin
      if (time_bit.write_timestamp_que = NIL) or
         (timestamp <= time_bit.write_timestamp_que^.stamp)
      then
        begin
          send_to_data_manager(request);
          remove_time_stamp(timestamp,time_bit,R);
          check_wait_queue(R);
        end
      else
        update_time_stamp(timestamp,time_bit,R);
    end;
  end;

  PROCEDURE write_protocol ;
      { (request : trans_request; timestamp : integer) }
  begin
    with colours^ do begin
      if ((time_bit.write_timestamp_que = NIL) or
          (timestamp <=
            time_bit.write_timestamp_que^.stamp)) and
         ((time_bit.read_timestamp_que = NIL) or
            (timestamp
               <= time_bit.read_timestamp_que^.stamp))
      then
        begin
          send_to_data_manager(request);
          remove_time_stamp(timestamp,time_bit,W);
          check_wait_queue(W);
        end
      else
        update_time_stamp(timestamp,time_bit,W);
    end;
  end;

begin
  find_object(request.access_key);
  case request.access of
    R : read_protocol(request,timestamp);
    W : write_protocol(request,timestamp);
  end;
  update(colours);
end;

begin
  initialization;
  repeat
```

```
    get_next_request(request);
    timestamp := transaction_timestamp(request.id);
    if request.access = P
    then claim_all_object(request.id,timestamp)
    else process_request(request,timestamp);
  until eof(timefile);
end.
```

Appendix E
Sample Test Runs

In this appendix, four different schedules are being used as the input schedules to the mechanisms presented earlier. This is to demonstrate how these mechanisms reshuffle the submitted requests into a correct schedule; also , to show that the algorithms developed in chapter 4 are correct.

For simplicity, we assume that there are only four records in the 'colours' database. These four records are :

jane	BLUE
jenny	RED
jerry	GREEN
jim	YELLOW

The four input schedules are :

1. Schedule **S1** -

   S1 = 1 R jenny
   2 R jenny
   1 W jenny
   2 W jenny

2. Schedule **S2** -

   S2= 1 R jenny
   2 R jenny
   1 W jenny
   1 R jim
   1 W jim
   2 R jim

3. Schedule **S3** -

   S3= 1 R jenny

2 R jenny

1 W jenny

2 W jim

3 R jim

2 W jenny

4. Schedule **S4** -

S4= 1 R jenny

1 W jenny

2 R jenny

2 W jenny

The first three schedules will result in various concurrency control problems and therefore required to be rescheduled. The purpose of including the last schedule which is a serial schedule is to show that the mechanisms do recognize it as a correct schedule.

Dynamic Two-Phase Locking Mechanism

1. Output schedule for S1.

```
          1 Rjenny
          2 Rjenny
cycle ->          1
cycle ->          2         1
deadlock detected !!!
abort transaction 2
          1 Wjenny
```

Deadlock occurs in this schedule and is resolved by aborting transaction 2. The abortion of transaction 2 also avoid the loss of transaction 1's update on *jenny*'s record.

2. Output schedule for S2.

```
          1 Rjenny
          2 Rjenny
          2 Rjim
          1 Wjenny
          1 Rjim
          1 Wjim
```

Notice that request (1 W jenny) from transation 1 is delayed till transaction 2 terminates. This avoids the problem of inconsistent retrieval of *jim*'s and *jenny*'s records.

3. Output schedule for S3.

```
          1 Rjenny
          2 Rjenny
          2 Wjim
cycle ->          1
cycle ->          2         1
deadlock detected !!!
abort transaction 2
          1 Wjenny
          3 Rjim
```

Deadlock occurs in this schedule and is resolved by aborting transaction 2. Notice that transaction 3's request to read *jim*'s record is delayed. This avoids transaction 3 from reading the 'phantom' record updated by transaction 2.

4. Output schedule for S4.

```
          1 Rjenny
          1 Wjenny
          2 Rjenny
          2 Wjenny
```

S4 is output without any rescheduling.

Pre-Transaction Two-Phase Locking Mechanism

1. Output schedule for S1.

```
All locks requested by transaction 1 granted.
Signal to send rest of transaction !!!

          1 Rjenny
          1 Wjenny
All locks requested by transaction 2 granted.
Signal to send rest of transaction !!!

          2 Rjenny
          2 Wjenny
```

S1 is rescheduled into a serial schedule with transaction 1 first then transaction 2. This avoids the problem of loss of update.

2. Output schedule for S2.

```
All locks requested by transaction 1 granted.
Signal to send rest of transaction !!!

          1 Rjenny
          1 Wjenny
          1 Rjim
          1 Wjim
All locks requested by transaction 2 granted.
Signal to send rest of transaction !!!

          2 Rjenny
          2 Rjim
```

Again, the output serial schedule avoids the problem of inconsistent retrieval of *jim*'s and *jenny*'s records.

3. Output schedule for S3.

```
All locks requested by transaction 1 granted.
Signal to send rest of transaction !!!

          1 Rjenny
          1 Wjenny
All locks requested by transaction 2 granted.
Signal to send rest of transaction !!!

          2 Rjenny
          2 Wjim
          2 Wjenny
All locks requested by transaction 3 granted.
Signal to send rest of transaction !!!
```

```
            3 Rjim
```

4. Output schedule for S4.

```
All locks requested by transaction 1 granted.
Signal to send rest of transaction !!!

            1 Rjenny
            1 Wjenny
All locks requested by transaction 2 granted.
Signal to send rest of transaction !!!

            2 Rjenny
            2 Wjenny
```

S4 is output without any rescheduling.

Dynamic Timestamping Mechanism

1. Output schedule for S1.

```
          1 Rjenny
          2 Rjenny
Abort transaction 1 !!!
          2 Wjenny
```

Transaction 1 is aborted since its timestamp is older than the read-timestamp of *jenny*'s record. The abortion also avoid the loss of update problem.

2. Output schedule for S2.

```
          1 Rjenny
          2 Rjenny
Abort transaction 1 !!!
          2 Rjim
```

The abortion of transaction 1 in this example avoids the inconsistent retrieval of *jim*'s and *jenny*'s records.

3. Output schedule for S3.

```
          1 Rjenny
          2 Rjenny
Abort transaction 1 !!!
          2 Wjim
          3 Rjim
          2 Wjenny
```

Notice that transaction 3 reads *jim*'s record which has been updated by transaction 2 before transaction 2 terminates. If transaction 2 is aborted before termination, then transaction 3 is reading a 'phantom record'. (*See Section 3.3.3*)

4. Output schedule for S4.

```
          1 Rjenny
          1 Wjenny
          2 Rjenny
          2 Wjenny
```

S4 is output without any rescheduling.

Pre-Transaction Timestamping Mechanism

1. Output schedule for S1.

```
1 Rjenny
1 Wjenny
2 Rjenny
2 Wjenny
```

A serial schedule with transaction 1 completed first then transaction 2. This avoids the problem of loss of update.

2. Output schedule for S2.

```
1 Rjenny
1 Wjenny
2 Rjenny
1 Rjim
1 Wjim
2 Rjim
```

Transaction 2's request to read *jenny*'s record is delayed till tranaction 1 updated the record. This avoids the inconsistent retrieval of *jim*'s and *jenny*'s records. However, reading of phantom record may occur if transaction 1 aborts before termination. (*See Section 3.3.3*)

3. Output schedule for S3.

```
1 Rjenny
1 Wjenny
2 Rjenny
2 Wjim
3 Rjim
2 Wjenny
```

Again, if transaction 2 is aborted before termination, then transaction 3 is reading a 'phantom record'. (*See Section 3.3.3*)

4. Output schedule for S4.

```
1 Rjenny
1 Wjenny
2 Rjenny
2 Wjenny
```

S4 is output without any rescheduling.