Introduction

Single-User System: At most one user at a time can use the system.

Multiuser System: Many users can access the system concurrently.

Concurrency:

- Interleaved processing: concurrent execution of processes is interleaved in a single CPU
- Parallel processing: processes are concurrently executed in multiple CPUs.

Concurrency Control, Recovery Mechanisms

Transactions - states, properties; Schedules

Concurrency Control - problems, approaches (locking, timestamping)

Recovery - problems, recovery mechanisms

Transactions - introduction

A transaction: logical unit of database processing that includes one or more access operations (read - retrieval, write - insert or update, delete).

A transaction (set of operations) may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.

Transaction boundaries: Begin and End transaction.

An application program may contain several transactions separated by the Begin and End transaction boundaries.

Reading involves:

finding address of a disk block that contains the item X

copying that disk block to a buffer

copying item X from buffer to program variable X.

Writing involves:

Find the address of the disk block that contains item X.

Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

Copy item X from the program variable named X into its correct location in the buffer.

Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Sample Transaction

read_item(X)

X : = X - N

write_item(X)

Concurrency Control

In most DBMS environments, it is desirable to allow many people to access the database at the same time.

Hence, many transactions running at once.

Needed to overcome problems that will arise if we allow unchecked access to the database.

The Lost Update Problem:

```
   T1                                   T2

read_item(X);

X := X-N;

                                    read_item(X);

                                    X := X+M;

write_item(X);

                                   write_item(X);
```

This results in the `incorrect' value being stored.

Temporary Update Problem:

```
   T1                                  T2

read_item(X);

X := X-N;

write_item(X);

                                   read_item(X);

                                   X := X+M;
                                   write_item(X);

read_item(Y);

    .

    .

 <CRASH>
```

Recovery mechanism will undo the effect of T1; the value of X will be changed back; T2 has the`incorrect' values

Incorrect Summary Problem

occurs when one transaction is calculating a sum (or some other aggregate function) of a range of values and another transaction is concurrently changing those items.

We need means to prevent these types of problems occurring.

Exercise: Draw a sample schedule that shows the incorrect summary problem.

Recovery

If a transaction is submitted to the DBMS, the system should ensure that either:

the transaction is completed successfully and it's effect recorded *or*

the transaction fails and has no effect on the database.

Partial execution of a transaction should *not* occur

Transactions can fail for a variety of reasons:

- System Crash
- Transaction Error
- Exception Conditions
- Concurrency Control Enforcement
- Disk Error
- Catastrophes

Main operations of a transaction

begin_transaction

read_item or write_item

end_transaction

commit

rollback (a transaction)

Undo (an operation)

Redo (an operation)

States of a transaction

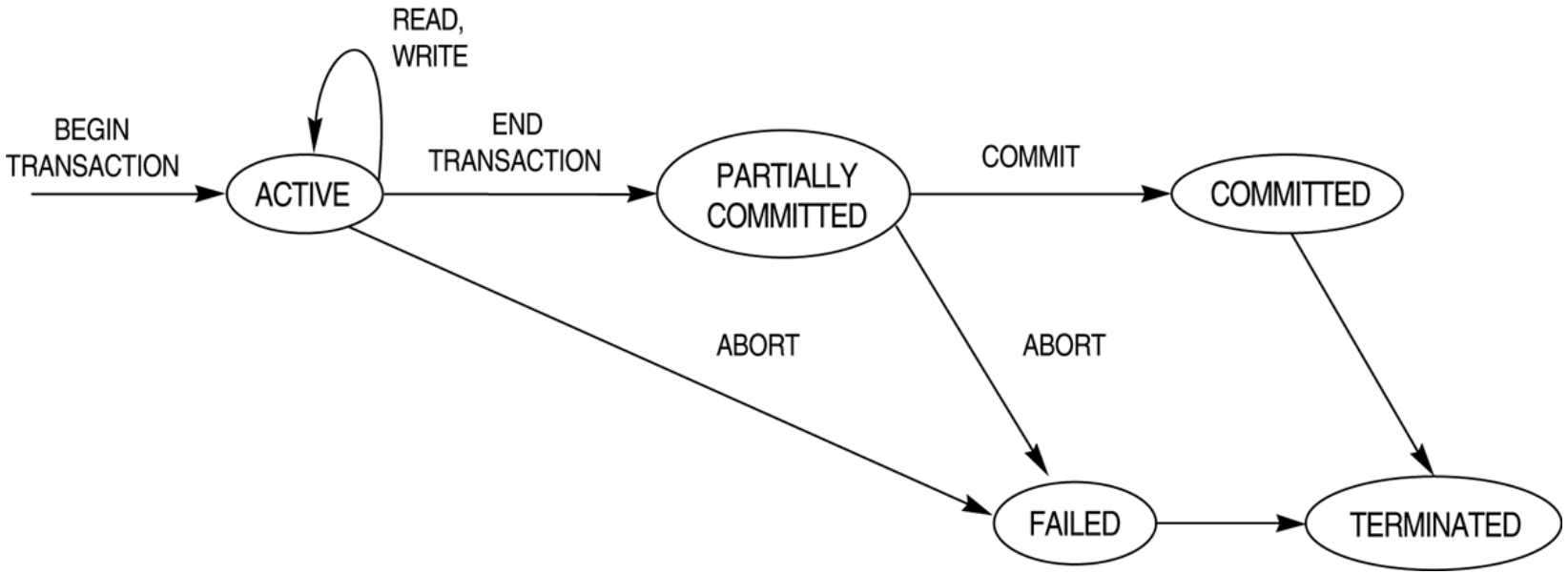
Active state

Partially committed state

Committed state

Failed state

Terminated State


READ,
WRITE
BEGIN
TRANSACTION
ACTIVE
END
TRANSACTION
PARTIALLY
COMMITTED
COMMIT
COMMITTED
ABORT
ABORT
FAILED
TERMINATED

System Log

A *system log or journal* is usually maintained by the DBMS in order to facilitate recovery

The following operations for each transaction are recorded.

System Log

```
start_transaction, T
write_item, T, X, old_value, new_value
read_item, T, X
commit, T
```

Commit Point

A transaction reaches its commit point if:

It finishes successfully

effects are recorded in log

Following a commit point of a transaction any updates by that transaction are considered to permanently stored in the database

A *{commit, T}* entry is recorded in the log

Desirable Properties of transactions:

Atomicity: a transaction should be performed completely or not at all

Consistency Preservation: a transaction should take the database from one consistent state to another

Isolation: updates of a transaction T should not be visible to other transactions until T commits.

Durability: updates made by a committed transaction should not be undone later due to failure.

These four properties are often referred to the ACID properties of a transaction.

Serializability

A schedule is any collection of transactions T_1, T_2, … , T_N). Each transaction can contain a number of read and write operations.

A desirable property of a schedule is that it is serializable.

A serial schedule is a schedule such that there is no interleaving of the operations of the transactions

If a schedule is serial we can guarantee that no lost updates, incorrect summary problems etc. will arise

One potential means to enforce concurrency control is to allow only serial schedules

However, this is far too limiting a constraint and would severely limit the throughput of the system

Ideally, we wish to allow interleaving of operations but maintain `equivalence' to a serial schedule.

A schedule that is `equivalent' to a serial schedule is known as a *serializable schedule.*

Need to define `equivalence' of schedules. The most commonly adopted definition is that of *conflict equivalence*.

Defn: Conflicting operations: 2 operations are said to conflict if (i) they access the same item and (ii) at least one of these operations is a *write.*

We say there is a conflict between two transactions T_1 and T_2 if they contain operations that conflict with each other.

A schedule S is said to be *conflict serializable* if the conflicting operations occur in exactly the same order as in some serial schedule

Given a schedule S of transactions (T_1, … T_N) we can test for conflict serializability using the following algorithm:

for each transaction create a node

Create an edge between node T1 and T2 if:

i) T1 issues read_item(X) before

T2 issues write_item(X) *or*

ii) T1 issues write_item(X) before

T2 issues read_item(X) *or*

iii) T1 issues write_item(X) before

T2 issues write_item(X)

If a cycle exists => not conflict-serializable

else conflict-serializable

Consider again the schedule we had to illustrate the lost update problem.

Graph contains a cycle, hence not serializable

If a cycle exists => not conflict-serializable

else conflict-serializable

This method of checking for conflict serializability is not practical in real world scenarios, as we do not know:

which transactions will be run

which operations they will contain.

We need to develop techniques that will guarantee conflict-serializability. We need to reject any operation that violates the principles of conflict serializability

The two main approaches are:

- locking protocols
- time-stamping