TRANSACTION MANAGEMENT - 1

U08049 Database Design – 2013

Learning outcomes

- Understand the nature of a transaction in a database system
- Explain the need for concurrency, and the problems to be overcome
- Describe the causes of database failure, and the process of recovery

Transactions

- An action, or series of actions, applied to a database, which must be considered as non-decomposable (i.e. atomic).
- The ‘consideration’ is for the integrity of the database – see the following examples

Borrowing a book from a library

MEMBER

Member_id	name	Current_limit
p0076635	smith	3
p8863443	khan	5

COPY

isbn	Copy#	On_loan_to
00-987-2	1	
00-987-2	2	

BOOK

isbn	title	Copies_available
00-987-2	Using UML	3

SQL code to borrow a book

```
update member set current_limit = current_limit - 1
            where member_ID = ‘p0076635’;
update copy set on_loan_to = ‘p0076635’
            where isbn = 00-987-2
            and copy# = 1;
update book set copies_available = copies_available - 1
            where isbn = 00-987-2;
```

- These three SQL statements must all be executed – or none at all.
- Note that this code does not include any ‘application’ code – for simplicity

SQL code to increase borrow limits

```
update member set current_limit = current_limit + 4 ;
```

- This SQL statement updates all members’ borrow limit
- If the database fails part way through the updates, some members may be updated twice, or some not at all.

Race Conditions

- A race condition or race hazard is a flaw in a system or process whereby the output and/or result of the process is unexpectedly and critically dependent on the sequence or timing of other events.
- The term originates with the idea of two signals racing each other to influence the output first.
- Race conditions can occur in all electronics systems, especially logic circuits, and in computer software, especially multithreaded or distributed programs.

A simple intuitive example

- Suppose the Prime Minister is signing papers:
  - He first looks at each paper, then signs it.
  - Every now and then, the phone rings and the PM is distracted by another task.
  - While the PM is busy with another task, an evil Civil Servant exchanges the paper he has just looked at but not signed yet.
  - The PM will sign the new paper without having looked at it, passing a law declaring software vulnerability research a criminal offence.
- In this case, the solution is obvious: do not let yourself be interrupted between looking and signing.

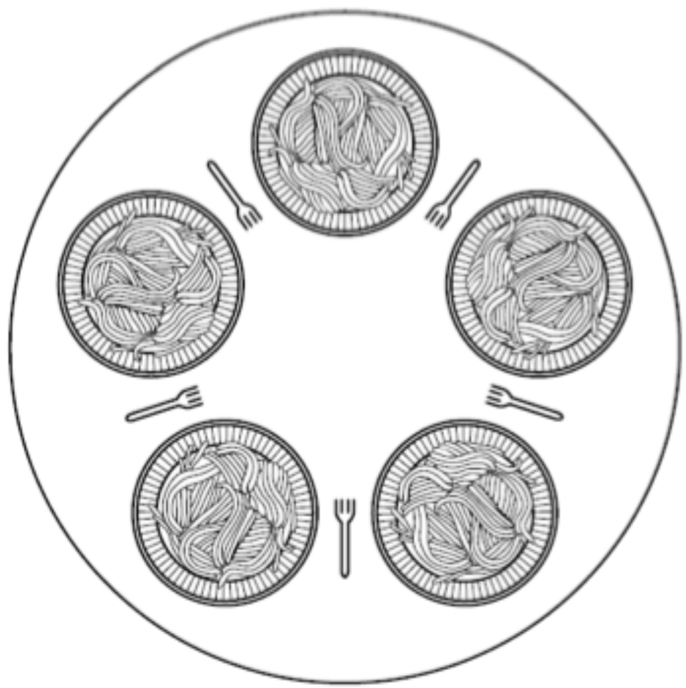
Race Conditions

A race condition may (will eventually) occur if:

- A process A performs a sequence of operations accessing a shared resource R, and
- the security depends on these operations not being interrupted by another, concurrent process B accessing the same resource R, and
- A does not ensure that its operations are atomic (i.e. non-interruptible).

General statement of the Race Condition problem

The dining philosophers problem:
Lunch time



The Dining Philosophers problem

- A non-solution to the problem:

```
#define N 5                              /* number of philosophers */

void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                        /* philosopher is thinking */
        take_fork(i);                    /* take left fork */
        take_fork((i+1) % N);            /* take right fork; % is modulo operator */
        eat( );                          /* yum-yum, spaghetti */
        put_fork(i);                     /* put left fork back on the table */
        put_fork((i+1) % N);             /* put right fork back on the table */
    }
}
```

The Dining Philosophers Problem

- A proper solution to the problem requires testing before access using a state machine (1 of 3 slides):

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N−1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */
```

The Dining Philosophers Problem

- Solution (2 of 3 slides)

```
void philosopher(int i)                    /* i: philosopher number, from 0 to N–1 */
{
      while (TRUE) {                       /* repeat forever */
            think( );                      /* philosopher is thinking */
            take_forks(i);                 /* acquire two forks or block */
            eat( );                        /* yum-yum, spaghetti */
            put_forks(i);                  /* put both forks back on table */
      }
}

void take_forks(int i)                     /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                        /* enter critical region */
      state[i] = HUNGRY;                   /* record fact that philosopher i is hungry */
      test(i);                             /* try to acquire 2 forks */
      up(&mutex);                          /* exit critical region */
      down(&s[i]);                         /* block if forks were not acquired */
}
```

The Dining Philosophers Problem

- Solution (3 of 3 slides)

```
void put_forks(i)                        /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                        /* enter critical region */
    state[i] = THINKING;                 /* philosopher has finished eating */
    test(LEFT);                          /* see if left neighbor can now eat */
    test(RIGHT);                         /* see if right neighbor can now eat */
    up(&mutex);                          /* exit critical region */
}

void test(i)                             /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Time-of-check-to-time-of-use

A time-of-check-to-time-of-use bug (TOCTTOU - pronounced "TOCK too") is a software bug caused by changes in a system between the checking of a condition (such as a security credential) and the use of the results

- Time-of-check-time-of-use ("TOCTOU") errors of that check.
  - Process A checks a security-relevant property of resource R,
  - Process B concurrently changes that property of R,
  - Process A uses R based on the results of its previous check.
- Concurrent updates to a database table
  - Process A: if (condition for field 1) then do something to field 2
  - However process B changes field 1 concurrently
  - Result: invalid combination of field values!

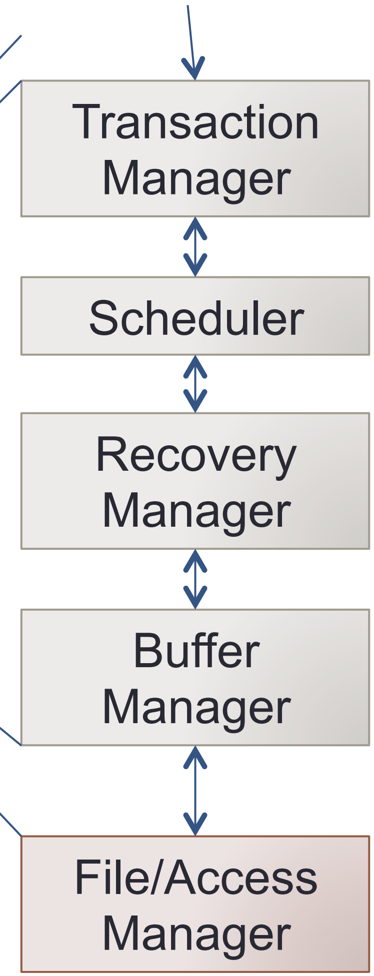
Database Management Systems

APPLICATION LAYER – programs, queries, etc. run in main memory

DBMS– provides support for transactions, including scheduling and recovery

OPERATING SYSTEM – providing file management and access management of secondary storage

SECONDARY STORAGE– stores data and system catalog PERMANENTLY.

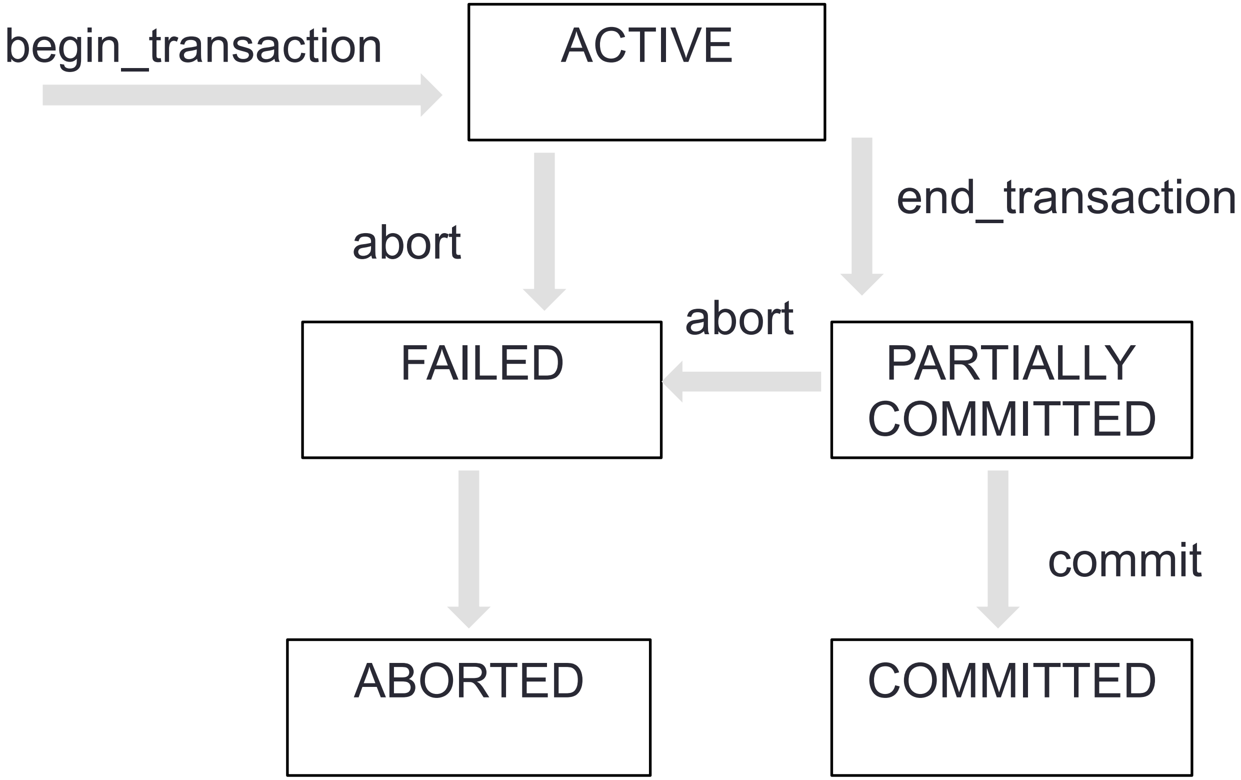


Properties of transactions (ACID)

- Atomicity – A transaction is a non-decomposable piece of work, that is either executed* in its entirety, or not at all
- Consistency – a transaction changes a consistent state of the stored database to another consistent state
- Isolation – partial effects of a transaction should not be visible to other transactions
- Durability – the effects of successful transactions are permanently recorded

* Executed, here, means that its results have been stored in secondary storage

States of a Transaction



Any ABORTed transaction may need to be ROLLED BACK or UNDONE, if some of its results are permanently stored.

Commercial databases

- A key aim of developing a database is to provide access to corporate data to many people.
- Key parts of the database may be duplicated and distributed, but this adds to complexity (and will not be dealt with in this course).
- Users will expect (apparently) simultaneous access to the same database.
- The DBMS manages multiple, concurrent transactions – maintaining Isolation

Possible problems caused by concurrency

- The following examples are based on a bank account system
- Transactions are submitted by multiple (possible remote) systems, but are dealt with by one DBMS, with one permanent version of the data.
- One way of dealing with concurrency would be to present the transactions to the DBMS, one at a time, and wait for each to complete!
- Concurrency implies maximising parallel processing – one CPU can deal with many transactions, leaving I/O to other processors.

The Lost Update Problem

time	T1	T2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

Note: SQL statements have been reduced to insert, update, operations – or read and write

The Uncommitted Dependency Problem

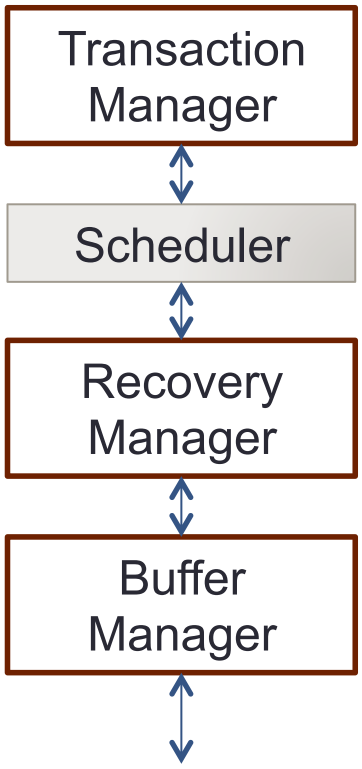
time	T3	T4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	...	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

The Inconsistent Analysis Problem

time	T5	T6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	$sum = sum + bal_x$	90	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	$sum = sum + bal_y$	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		$sum = sum + bal_z$	90	50	35	185
t_{11}		commit	90	50	35	185

The Scheduler

- A Schedule (of operations) is any interleaving of the operations of a set of transactions, that preserves the order of the operations within individual transactions;
- A Serial Schedule is the (guaranteed safe) schedule with no interleaving
- A Serializable Schedule is one which produces the same results as some Serial Schedule



Achieving Serializability

- Pessimistic approaches, delay transactions in case they cause problems:
  - Locking
  - Time stamping
- Optimistic approaches rely on recovery from problems, assuming that such problems are rare.

Locking

- Managed by the scheduler which grants locks to transactions for ‘part’ of the database.
  - May be a table, or rows/columns, or cells
  - Referred to here as a data item
- <u>Shared lock</u> – granted to any transaction needing to read a data item
- <u>Exclusive lock</u> – granted to the first transaction needing to write to a data item
- An exclusive lock will only be granted when there are no other locks on the data item
- Once an exclusive lock has been granted, no other transactions will be granted either lock on the data item

Serializability and the Two-phase Locking Protocol

- In order to guarantee serializability, the scheduler must enforce the following protocol:

  - A lock is required by a transaction if it needs to read or write to the data item;
  - A transaction must acquire all required locks before it releases any.
  - Once a transaction releases a lock, it can never acquire any new locks.

- This reduces concurrency, but prevents transactions interfering with one-another

Preventing the Lost Update Problem

time	T1	T2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	Commit/unlock(bal_x)		190

Preventing the Uncommitted Dependency Problem

time	T1	T2	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Preventing the Inconsistent Analysis Problem

time	T5	T6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(all)	90	50	35	175

Summary

- Transactions are fundamental to database management
- Serializable schedules balance concurrency with consistency
- One method for achieving serializability is locking
- You have seen how the two-phase locking (2PL) protocol prevents consistency problems.