Transactions on persistent data

We have seen how to make a single operation on a shared data object in main memory ATOMIC (indivisible) by enforcing its execution under mutual exclusion

Now consider how to implement a single atomic operation on persistent data

Note that on a crash, all data in main memory is lost – *what reached the disc before the crash?*

- concurrency control can be implemented as before, enforcing mutual exclusion
- the new problem is how to achieve atomicity in the presence of crashes
- i.e. the operation has externally visible effects and the crash may occur at any time during the operation

Definition: ATOMIC operation:
- if an atomic operation terminates normally, all its effects are made permanent
- else it has no effect at all

e.g. ***credit (account #, £1000)***

- note: tell the user "done" AFTER checking that the new value has been written

Crash model, idempotent operations and atomicity

We shall assume that a crash is fail-stop:
processors, TLBs, caches, main memory are lost
persistent memory on disc is not lost – *but what state is it in?*

To what extent can operations be made idempotent (repeatable)?
e.g. ***append-to-file (address-in-memory, byte-count)*** is not
e.g. ***append-to-file (address-in-memory, byte-count, position in file)*** is repeatable
- but the system may use an implicit pointer (e.g. UNIX)
- in general, not every operation can be made repeatable

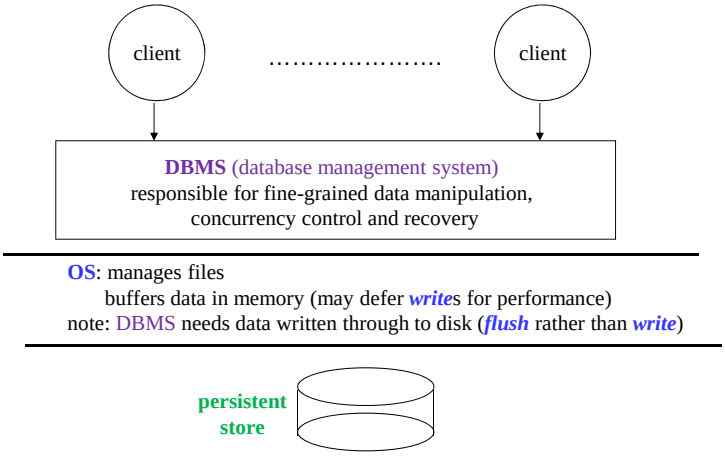
How can atomic operations on persistent data be implemented?
- logging: update the data in place,
  but ***first*** write a separate log record to disc of the old and new values
  on a crash we can use the log record to roll-back or forward

- shadowing: keep the old data intact
  build up a new version of the data
  flip atomically from the old to the new version, e.g. flip a pointer

in both cases output "done" to the client *after* committing the update.

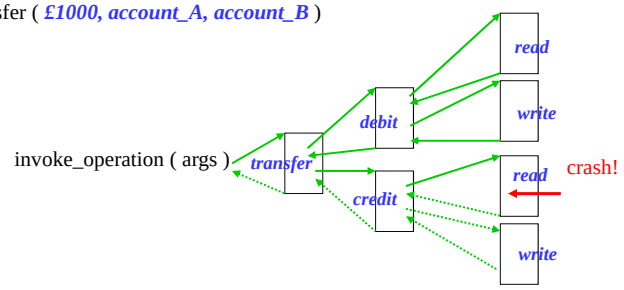
Atomic operations involving persistence – system components

A typical structure of a centralised transaction processing system



Transactions – composite operations on persistent objects

One operation on shared persistent data atomic in the presence of concurrency and crashes – OK!
Now suppose a meaningful operation is composite, comprising several such operations:
e.g. delete a file (remove link from directory, remove metadata, add file blocks to free list)
e.g. transfer (*£1000, account_A, account_B*)



Concurrency control: why not lock all data – do all operations – unlock?
But contention may be rare, and "locking all" may impose overhead, slow response.
Concurrency gives better performance but ***problems can occur – see next slides.***

Crashes: have any permanent/visible/persistent changes been made to any of the data?
Has an ***inconsistent state*** resulted from the crash?

Composite operations with no concurrency control the "lost update" problem

What is defined as a single operation on persistent data?
In the example below, ***read*** and ***write*** to disc are taken to be separate operations.
This is known as read/write semantics (the lowest level for transactions – we can do better)

transaction P: ***transfer (£1000, account_A, account_B)***

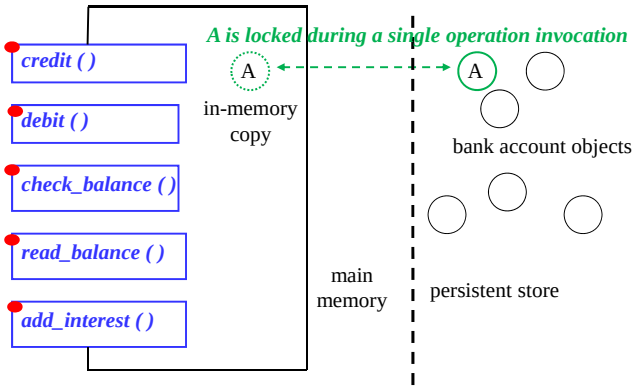
transaction Q: ***transfer (£200, account_C, account_A)***

Transfer operations may execute correctly until an unfortunate interleaving occurs:

```
                                        debit ( £200, account_C ) :
                                            read ( account_C )  \\ into memory
                                            account_C = account_C - 200
                                            write ( account_C) \\ to disc
                                        // Q has debited account_C by £200
debit ( £1000, account_A ) :
    read ( account_A ) \\ into memory
    account_A= account_A - £1000
                                        credit ( £200, account_A ) :
                                            read ( account_A ) \\ into memory
                                            account_A= account_A + £200
    write ( account_A) \\ to disc
                                            write ( account_A) \\ to disc
// P's update to account_A is lost      // Q's update to account_A overwrites P's update.
```


Object semantics - 1

Define atomic operations on persistent objects – i.e. higher level object semantics
e.g. bank account objects, with operations that include ***credit*** and ***debit***, assumed atomic.
Omitting ***create*** and ***delete*** we might have:



Object semantics – 2

Object operations are atomic – we have object semantics, not read/write semantics.
Does this solve the concurrency control problems? NO

transaction P	transaction Q
transfer (£1000, account_A, account_B)	***add_interest (account_N)***

Suppose ***add_interest*** updates all accounts daily.
As before, the operations may execute correctly until an unfortunate interleaving occurs.

check_balance (£1000, account_A)	*\\ keeping A locked between operations*
debit (£1000, account_A)	*\\ doesn't solve the problem*
	add_interest (account_A)
	add_interest (account_B)
credit (£1000, account_B)	

The interest on £1000 is lost to the account holders, gained by the system.
The database state is (arguably) incorrect
The problem is due to the visibility of the effects of the sub-operations of ***transfer***,
involving related operations on two objects
transfer is the meaningful high-level operation – how to make a composite operation atomic?

Object semantics – 3

Make ***lock*** an explicit operation. Hold one lock while acquiring others?
Two-phase locking (2PL) is the most common method of database concurrency control

transaction P

transfer (£1000, account_A, account_B)

lock (account_A)
check_balance (£1000, account_A)
debit (£1000, account_A)

lock A held
lock B requested

lock (account_B)
credit (£1000, account_B)
unlock (account_B)
unlock (account_A

Transaction Q?
can't use 2PL on all objects in the system.
Should the service be unavailable
while interest is added to all objects?
NOT NEEDED!
The operations on the accounts are not related.
Q is NOT executing a huge composite operation
but many small individual-object transactions .
Transactions such as P prevent lost interest.

But recall the conditions for deadlock ……. We'll come back to this …….

Transactions – notation

Example using transaction identifiers, ***commit*** and ***abort***:

T_i = $start_i$,
    checkbalance $_i$ ***(account_A, £1000) , ……..***
    $debit_i$ ***(£1000, account_A) ,***
    $credit_i$ ***(£1000, account_B) ,***
    $commit_i$

Each operation of a transaction is tagged with the transaction identifier ***i***
The last operation on successful termination is ***commit***
If the transaction fails, e.g. ***checkbalance*** returns fail/false, the last operation is ***abort***
On ***abort*** any intermediate effects of the transaction must be UNDONE

Suppose a crash occurs after ***debit***. A crash ***abort***s an uncommitted transaction.
    account_A must be restored to its initial state
    (note that ***credit*** is the UNDO operation for ***debit***)

The ***abort*** operation could be given to the application programmer, e.g.:

```
transaction {
    if  checkbalance ( account_A, £1000 )
    then  transfer ( £1000, account_A, account_B); commit
    else  abort }
```


Serialisability – intuition (definition on slide 15)

If transactions execute **strictly serially** then the **system state** (and any output) is **correct**.
i.e. transactions are meaningful, high-level operations.
The execution of a transaction moves the system from one consistent state to another.

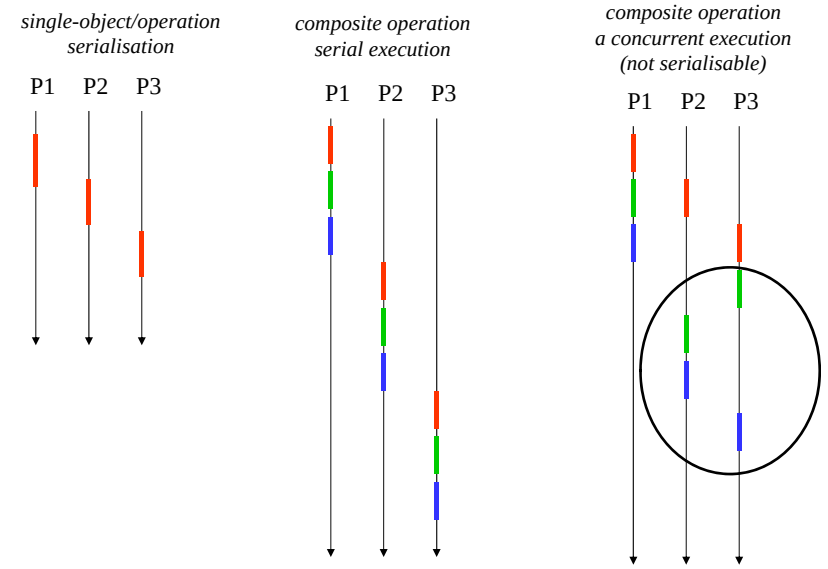
If we can show that a concurrent, interleaved execution is equivalent to some serial execution then the concurrent execution is correct

Examples:

serial execution:	non-serialisable execution:	serialisable execution:
debit (£1000, account_A)	***debit (£1000, account_A)***	***debit (£1000, account_A)***
credit (£1000, account_B)	***add_interest (account_A)***	***add_interest (account_A)***
add_interest (account_A)	***add_interest (account_B)***	***credit (£1000, account_B)***
add_interest (account_B)	***credit (£1000, account_B)***	***add_interest (account_B)***

serial execution	non-serialisable execution:	serialisable execution:
add_interest (account_A)	***add_interest (account_A)***	***add_interest (account_A)***
add_interest (account_B)	***debit (£1000, account_A)***	***debit (£1000, account_A)***
debit (£1000, account_A)	***credit (£1000, account_B)***	***add_interest (account_B)***
credit (£1000, account_B)	***add_interest (account_B)***	***credit (£1000, account_B)***

Serialisation of composite operations - visualisation



Transactions - ACID properties

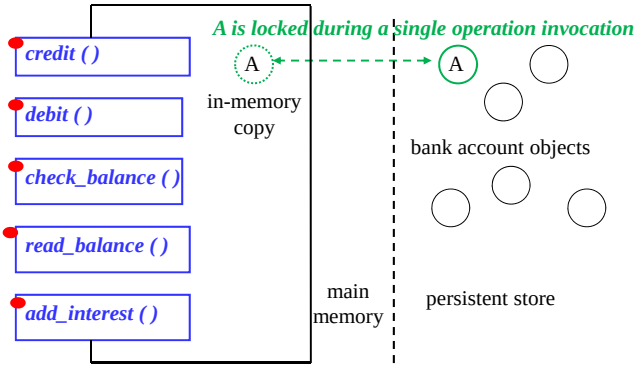
Atomicity	all or none of the operations are done (executed on the *persistent* store)
Consistency	a transaction transforms the system from one consistent state to another
Isolation	the effects of a transaction are not visible to other transactions until it is committed
Durability	the effects of a committed transaction endure/persist

C and **I** are defined with concurrency control primarily in mind,
A and **D** with requirements for crash recovery primarily in mind
But the mechanisms for enforcing concurrency control and crash recovery are related.

Strict enforcement of **I** reduces concurrency, sometimes unnecessarily.
See later: can **I** be relaxed in implementations while still ensuring serialisability?

D can be implemented by using techniques such as stable storage, involving redundant disc writes, RAID array techniques, etc. and we shall not study this property further

Object model for transaction processing – object semantics



- objects are identified uniquely
- each operation is atomic i.e. an object is locked during an operation invocation
- the object has a single clock
- for each operation invocation completed, the object records completion time and transaction-ID

Object semantics – definition of conflicting operations

DEFINITION of conflicting (non-commutative) operations
The **final** state or output value
depends on the **order** in which these operations are carried out

debit and ***add_interest*** conflict,
credit and ***add_interest*** conflict,

credit and ***credit*** or ***debit*** and ***debit*** or ***credit*** and ***debit*** do not conflict
so they can be applied to objects in any order

Arithmetic
+ and – do not conflict (are commutative) X +3+5 = X+5+3, X-2 +7 = X+7-2
* conflicts (does not commute) with + and –
X * 2 + 3 ≠ (X+3)*2

Serialisability is concerned with the **order** in which operations are invoked on objects.
We need only be concerned with conflicting, non-commutative operations.

Serialisability definition

For serialisability of two transactions it is necessary and sufficient for their order of execution of all conflicting pairs of operations to be the same for all the objects that are invoked by both

transaction T1
debit (£1000, account_A)

credit (£1000, account_B)

transaction T2
add_interest (account_A)
add_interest (account_B)

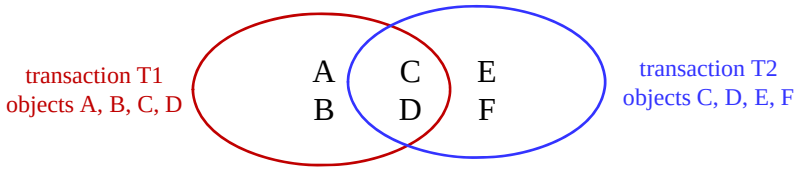
objects ***account_A*** and ***account_B*** are invoked by T1 and T2
operation ***add_interest*** conflicts with operations ***debit*** and ***credit***

object ***account_A*** T1 before T2

object ***account_B*** T2 before T1

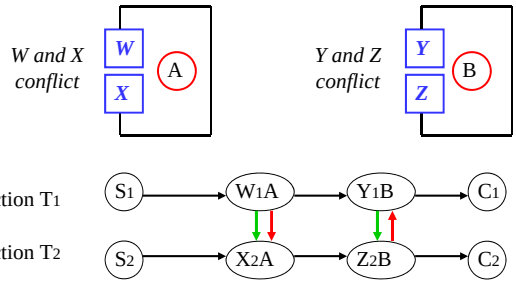
The above operation interleavings do NOT form a serialisable execution schedule

Visualisation of serialisability condition



Assuming T1 and T2 execute conflicting operations on C and D
A concurrent execution of T1 and T2 is serialisable if and only if
either T1 operates on both C and D before T2 $T_1 \longrightarrow T_2$
or T2 operates on both C and D before T1 $T_1 \longleftarrow T_2$

Serialisability – transaction execution representation



T1 and T2 are serialisable if both W1A is before X2A and Y1B is before Z2B
(or if both W1A is after X2A and Y1B is after Z2B)

T1 and T2 are NOT serialisable if W1A is before X2A and Y1B is after Z2B
(or if W1A is after X2A and Y1B is before Z2B)

Note that the **Isolation** property of transactions is not being enforced in the implementations.

Serialisation graphs

DEFINITION: A history represents the concurrent execution of a set of transactions.
(as in the previous slide when the order of execution of conflicting operations is included)

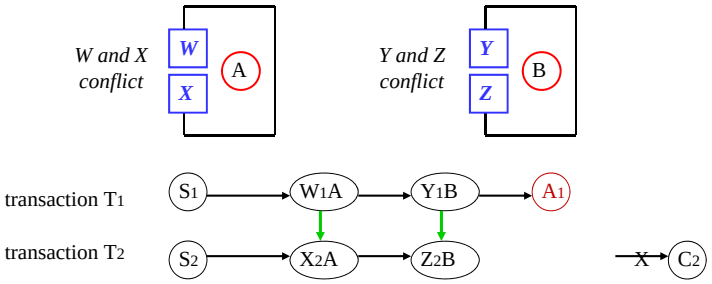
DEFINITION: A serialisable history represents a serialisable execution

DEFINITION: a serialisation graph shows only transaction IDs and dependencies between them.

T1 → T2 T1 ⇄ T2

A transaction history is serialisable if and only if its serialisation graph is acyclic

Cascading aborts



Suppose that to enforce serialisability the transaction scheduler makes T_2 execute conflicting operations on shared objects A and B after transaction T_1

Now suppose T_1 aborts after updating the objects
T_2 must also be aborted – a **CASCADING ABORT**

This has resulted from not enforcing the **Isolation** property of transactions.
T_2 has operated on uncommitted state (for maximum concurrency)
An execution in which **Isolation** is enforced is defined as **STRICT**

Recovering state – 1 (without conflicting operations)

To implement transactions, it must be possible to **recover** some **previously committed state**.
What are the implications of not enforcing the **Isolation** property?

	Money in accounts:	A	B	C
start1		£5000	£1000	£8000
credit1 (£1000, account_A)		£6000	…	…
credit1 (£500, account_B)		….	£1500	…
start2				
credit2 (£200, account_A)		£6200	…	…
credit1(£300, account_C)		…	…	£8300
abort1				
	undo			£8000
	undo		£1000	
	undo	£5200		

This is possible only because credits do not conflict and undo for ***credit*** is ***debit***

credit2 (£600, account_B)		….	£1600	…
abort2				
	undo		£1000	
	undo	£5000		

Recovering state – 2 (with conflicting operations)

		Money in account: A	
Start1		£5000	
credit1 (£1000, account_A)		£6000	
start2			
credit2 (£2000, account_A)		£8000	
start3			
add_interest (account_A)		£8008	
request commit ……. commit pended – state of uncommitted transactions has been used			
start4			
credit4 (£1000, account_A)		£9008	
request commit	***commit*** pended – state of uncommitted transactions has been used		
abort1	undo4	£8008	\\ *undo all to before conflicting operation*
	undo3	£8000	\\ *no need to undo T2s credit*
	undo1	£7000	
	redo3	£7007	
	redo4	£8007	
abort2	undo4	£7007	
	undo3	£7000	
	undo2	£5000	
	redo3	£5005	
	redo4	£6005	
commit3			
commit4			

Transactions - Summary

Considered composite operations on persistent objects subject to concurrency and crashes
looked at problems due to concurrent executions

Defined read/write semantics and object-operation semantics
which problems are solved, and which are not?

Defined conflicting (non-commutative) operations

Defined serialisability

Defined ACID properties of transactions

Looked at cascading aborts and recovering state

NEXT
Concurrency control and recovery