Transactions

In the next few lectures we motivate the need for transactions then study them in detail

From a single operation on a data object in a concurrent system, we extend to:
Composite operations: in main memory, and with persistent memory

We first study deadlock in general terms,
starting from composite operations in main memory, ref CCC 31, 32, 33

Then, continuing with single and composite operations:
Persistent data
Crashes
Atomic composite operations and how to implement them
Concurrency control with data in persistent memory
Serialisation concept to underpin transactions

Transactions: composite operations involving persistent data
Terminology
ACID properties
ACID properties; implications of relaxing isolation
serialisability, serialisation graphs
cascading aborts
recovering state

Deadlock

Systems that allocate resources dynamically are subject to deadlock.

We will encounter deadlock in transaction processing systems.

We now take some time to look at deadlock before returning to the development of transactions.

Recall: composite operations in main memory, ref CCC 32, an example of deadlock

Background policies that make deadlock possible, and what events make it occur dynamically?

Deadlock prevention – discussion of the conditions for avoidance and recovery.

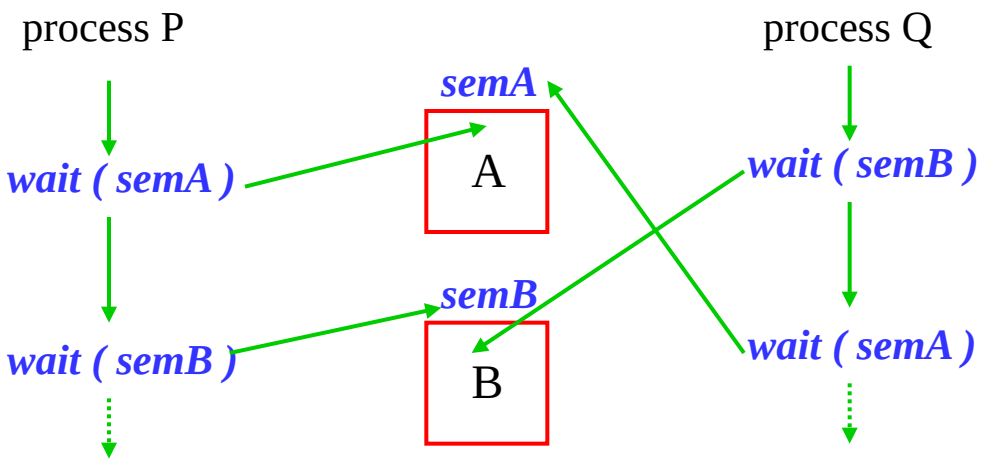
Dining philosophers program – example of deadlock and discussion of policies

Modelling deadlock – to support deadlock avoidance.
- object allocation, resource requests and cycle detection
- data structures and an algorithm for deadlock detection

Further reading

Composite operations with no concurrency control - 1

Recall the example below (CCC 32) involving only main memory
– we now highlight a condition for deadlock to exist:



At this point we have deadlock. Process P holds ***semA*** and is blocked, queued on ***semB***
Process Q holds ***semB*** and is blocked, queued on ***semA***
Neither process can proceed to use the resources and signal the respective semaphores.

A cycle of processes exists, where each holds one resource and is blocked waiting for another, held by another process in the cycle.

Conditions for deadlock *to exist*

1. Policy: mutual exclusion
   Processes can claim exclusive access to the resources they acquire

2. Policy: hold-while-waiting
   Processes can hold the resources they have already acquired while waiting for additional resources.

3. Policy: no pre-emption
   Resources cannot be forcibly removed from processes. Resources are explicitly released by processes (e.g. ***unlock/signal*** as above).

4. Dynamic occurrence: Circular wait (cycle)
   A circular chain of processes exists such that each process holds (at least) one resource being requested by the next process in the chain.

If ALL of the above hold then deadlock exists, if there is only one instance of each resource.
See 8, 10.

Other processes will be able to continue execution but the system is degraded by the resources held by the deadlocked processes.
Other processes may proceed to block on resources within the deadlock cycle.

Deadlock prevention

At all times at least one of the four conditions must not hold if deadlock is to be prevented by system design.

1 Policy: mutual exclusion
Cannot always be relaxed – introduced to prevent corruption of shared resources.

2. Policy: hold-while-waiting
Request all resources required in advance? Inefficient and costly.
Consider long-running transactions. Processes with large resource requirements could suffer starvation.

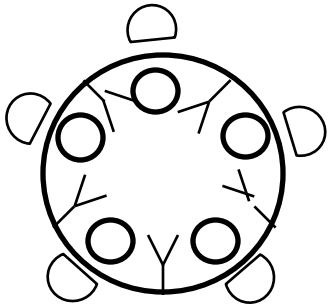
3. Policy: no pre-emption
Pre-emption could introduce the problems we will explore caused by visibility of intermediate results of transactions.

4. Dynamic occurrence: Circular wait (cycle)
Impose an order of use on resources – used by some OSs. Not easy to impose and check in general.

Perhaps allowing deadlock to occur, detecting and recovering by restarting some transactions is preferable.
NOTE – this (support for restart) may be in place for crash recovery.

The mechanisms for concurrency control and crash recovery could be combined.
We come back to this later. First, another example:

Dining philosophers (due to Dijkstra, 1965) - 1



Five philosophers spend their time thinking and eating. They each have a chair at a shared table with a shared bowl of food and shared forks – they need two forks to eat.
To eat they "execute" an identical algorithm –
pick up left fork, pick up right fork, eat, put down forks.

var fork : array [0 .. 4] of semaphore \\ all initialised to 1

philosopher i may then be specified as:

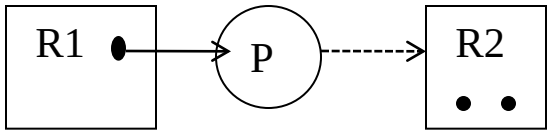
```
repeat
    wait ( fork [i] ) ;
    wait ( fork [i+1 mod 5 ] ) ;
    EAT
    signal ( fork [i] ) ;
    signal ( fork [i+1 mod 5 ] ) ;
    THINK
until false
```

Dining philosophers - 2

We have the policies in place for deadlock to be possible:
exclusive hold, hold-while-wait, no preemption.
Dynamically, deadlock can occur:
a cycle is created when the philosophers each acquire their left fork
and block waiting for their right fork.

The problem can be solved in a number of ways, essentially by ensuring
that at least one of the conditions necessary for deadlock to exist cannot hold
Breaking the symmetry of the algorithm can achieve this
e.g. make odds pick up their forks as specified, L then R,
and evens pick up their forks in reverse order, R then L.

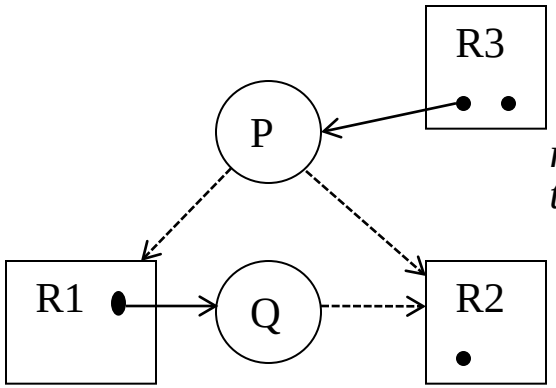
Object allocation and request – graphical notation



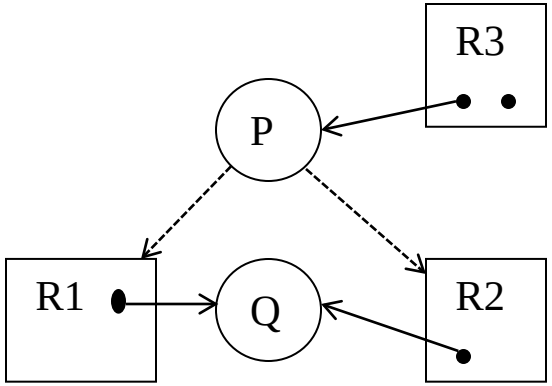
R1 and R2 are object/resource types. R1 has one instance and R2 has two.
The directed edge from the single instance of R1 to process P
indicates that P holds that resource.
The dashed directed edge from P to the object type R2 indicates that P has an outstanding request for an object of type R2.
P is therefore blocked, waiting for an R2.

If a cycle exists in such a graph and there is only one instance of each of the types involved in the cycle, then deadlock exists (necessary and sufficient condition).
If there is more than one object of some or all of the types, then a cycle is a necessary but not a sufficient condition for deadlock to exist.

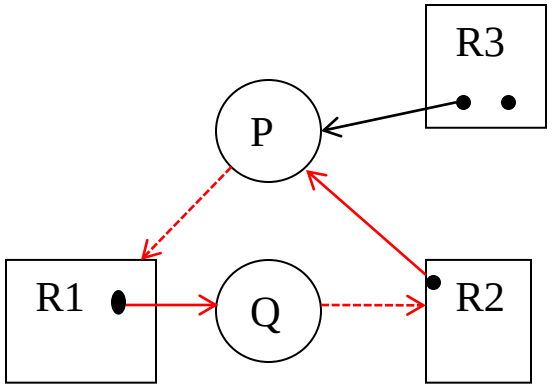
Dynamic object allocation and request – example



resource allocation must decide whether to give the R2 instance to P or to Q

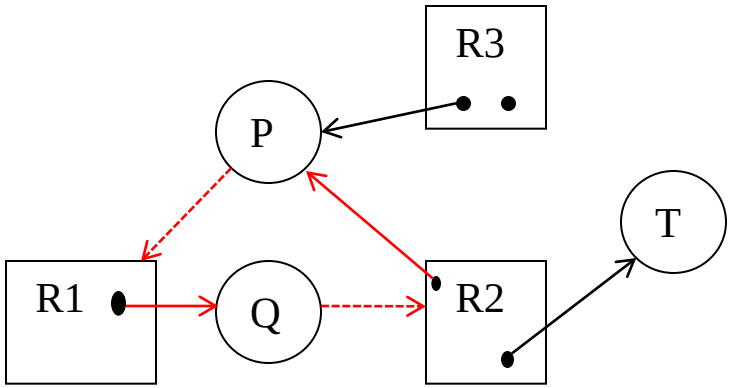


give the R2 instance to Q: no cycle
AFAIK, Q can complete and release R1 and R2, then P can have R1 and R2 and complete.
There may of course be further dynamic requests.

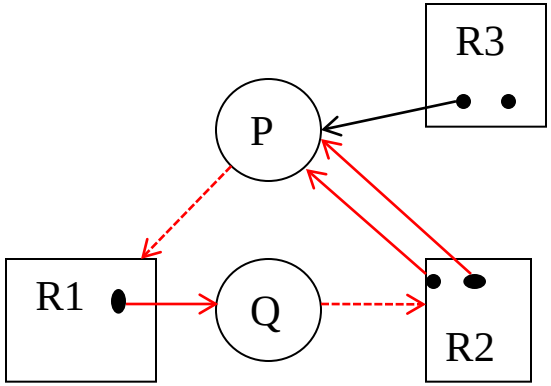


give the R2 instance to P: cycle = deadlock

Cycles without and with deadlock



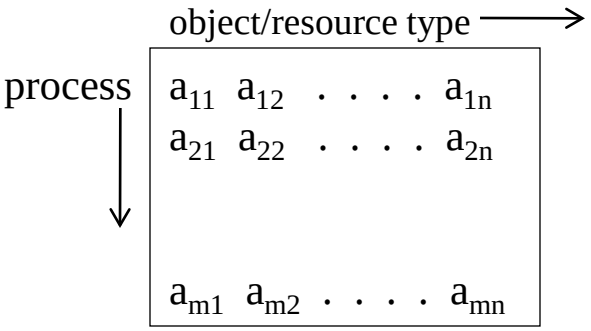
a cycle exists, but no deadlock
T could release R2, and unblock Q



a cycle and deadlock
P is blocked waiting for R1
Q is blocked, waiting for R2

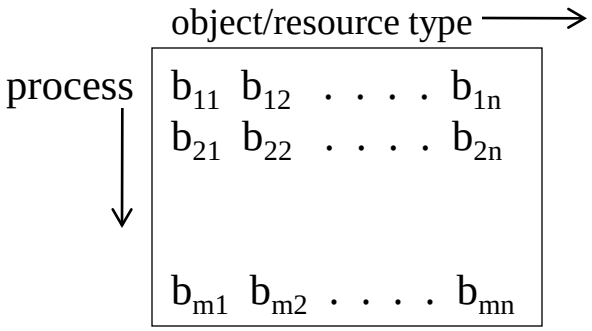
Data structures for resource/object allocation management

Allocation matrix A_{mn}



a_{ij} is the number of objects of type j allocated to process i

Request matrix B_{mn}



b_{ij} is the number of objects of type j requested by process i

objects being managed: $R_n = (r_1\, r_2\, \ldots\, r_n)$, the number of type i is r_i

objects available: $V_n = (v_1\, v_2\, \ldots\, v_n)$, the number of type i is v_i, computable from R_n minus the objects allocated

Algorithm for deadlock detection

Mark the rows of the allocation matrix that are NOT part of a deadlocked set

1. Mark all null rows of A (a process holding no resources cannot be part of a deadlocked set
2. Initialise a working vector W = V initially, the available objects
3. Search for an unmarked row, say row i, such that $B_i \leq W$
   (the objects that process i is requesting are “available” in W).
   If none is found, terminate the algorithm.
4. Set $W = W + A_i$ and mark row i. Return to step 3.

Example allocated: A	requested: B	total R	available V -> W initially
1 0 1 1 0	0 1 0 0 1	2 1 1 2 1	0 0 0 0 1
1 1 0 0 0	0 0 1 0 1		
0 0 0 1 0	0 0 0 0 1		
0 0 0 0 0	1 0 1 0 1		

Algorithm for deadlock detection - example

3. Search for an unmarked row, say row i, such that $B_i \leq W$
   If none is found, terminate the algorithm.
4. Set $W = W + A_i$ and mark row i. Return to step 3.

Example allocated: A requested: B total R available V -> W initially

A	B	total R	V -> W initially
1 0 1 1 0	0 1 0 0 1	2 1 1 2 1	0 0 0 0 1
1 1 0 0 0	0 0 1 0 1		
0 0 0 1 0	0 0 0 0 1		
0 0 0 0 0 X	1 0 1 0 1		

process 3's request can be satisfied

1 0 1 1 0
1 1 0 0 0
0 0 0 1 0 X
0 0 0 0 0 X

AFAIK process 3 can complete and return its resources

W becomes 0 0 0 1 1 (now "available")

processes 1 and 2 are deadlocked over objects 2 and 3

A	B
1 0 1 1 0	0 1 0 0 1
1 1 0 0 0	0 0 1 0 1
0 0 0 1 0 X	0 0 0 0 1
0 0 0 0 0 X	1 0 1 0 1

R = 2 1 1 2 1
W = 0 0 0 1 1

Deadlock – further reading

see Bacon “Concurrent Systems” or Bacon and Harris “Operating Systems”

- for a visualisation of the above algorithm showing the object allocations and requests

- for an extension of the approach for deadlock avoidance
  in the case where the maximum resource requests of all the processes are known statically
  But this turns out to be over-conservative ....

- If more information is available statically we might do better.
  In the case of multiphase processes, we know the order in which objects
  are released and requested.

- distributed deadlock detection, where the processes and objects reside on various
  nodes of a distributed system.

Persistent data

So far we have focussed on concurrency control for shared data in main memory.

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

Note that on a crash, all data in main memory is lost.

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

- concurrency control can be implemented as before

- 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

Definition: ATOMIC operation:

- if it terminates normally, all its effects are made permanent (stable storage abstraction)
- 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

To what extent can operations be made idempotent (repeatable)?
e.g. append-to-file (address-in-memory, amount of data) is not
e.g. append-to-file (address-in-memory, amount of data, position in file) is repeatable

- but the system may use an implicit pointer (e.g. UNIX)
- in general, not every operation can be made idempotent

How can atomic operations on persistent data be implemented?

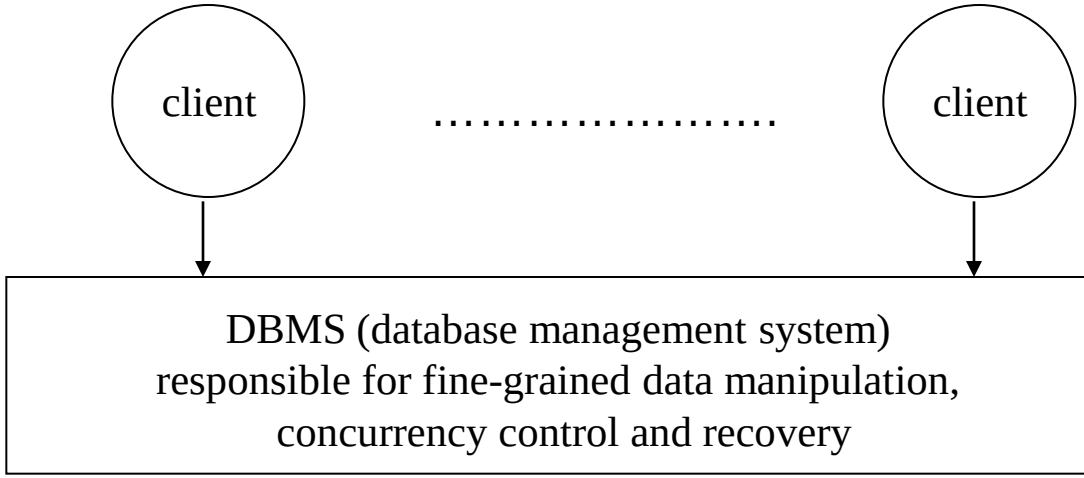
- logging: update the data in place,
  but *first* write a separate log record to disc of the old and new values
  on a crash can use these to roll-back or forward

- 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

on both cases output “done” to the client *after* committing the update.

Atomic operations involving persistence – system components

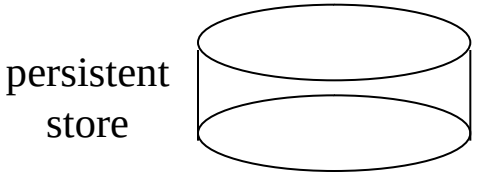
A typical structure of a centralised transaction processing system



OS: manages files

buffers data in memory (may defer ***write***s for performance)

note: DBMS needs data written through to disk (***flush*** rather than ***write***?)

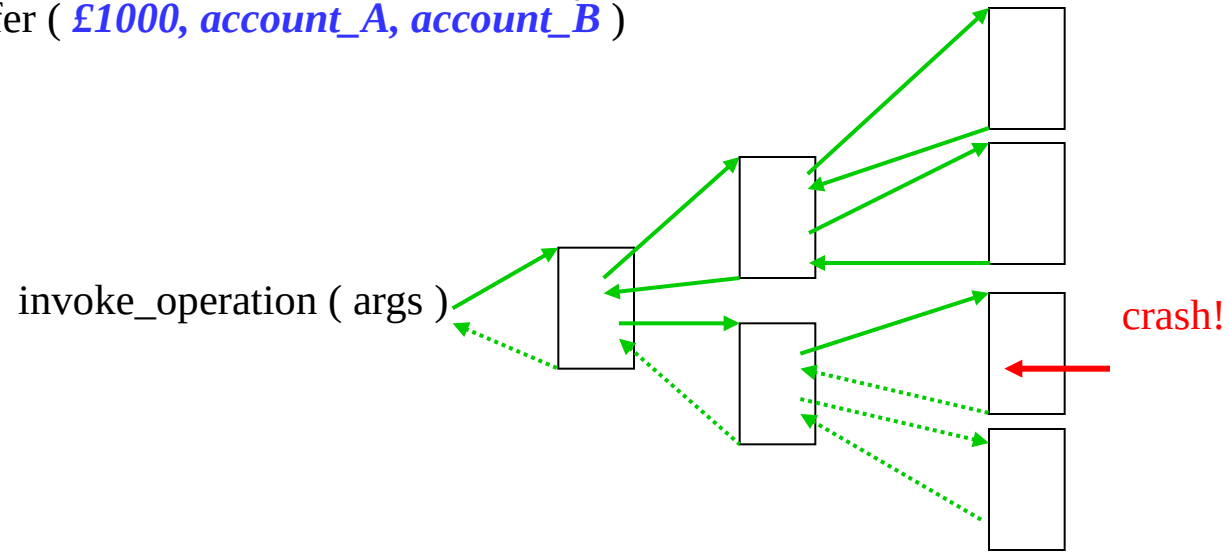


Introducing transactions – composite operations with persistence

We have studied how to make one operation on shared data atomic in the presence of concurrency and crashes.
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 and slow response.
Problems can occur if operations can be invoked concurrently – see next slides.

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

Composite operations with no concurrency control – 2
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.

process P

transfer (£1000, account_A, account_B)

process Q

transfer (£200, account_C, account_A)

As before, transfer operations may execute correctly until an unfortunate interleaving occurs:

debit (£200, account_C)
read (account_C)
write (account_C)

Q has debited account_C by £200

debit (£1000, account_A)
read (account_A)

credit (£200, account_A)
read (account_A)

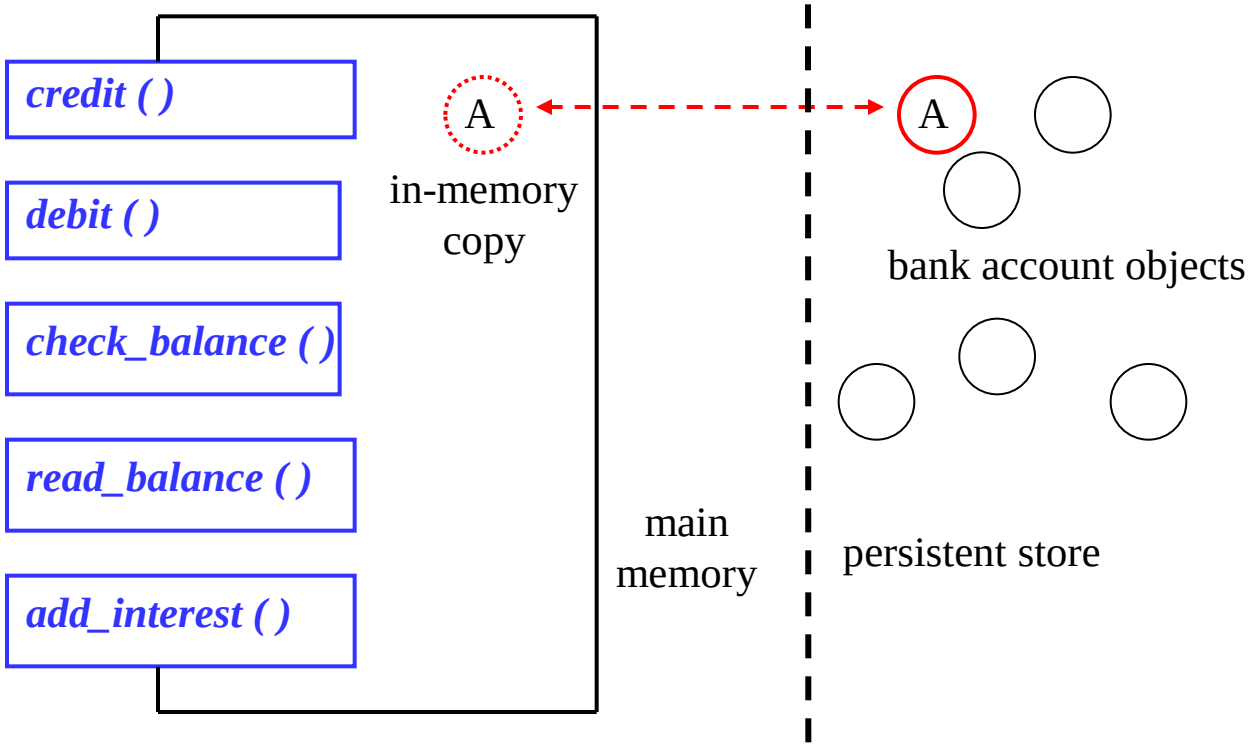
write (account_A)

write (account_A)

Q's update to account_A overwrites P's update.

Object semantics - 1

Define atomic operations on persistent objects e.g. bank account objects
with operations that include ***credit*** and ***debit,*** 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?

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

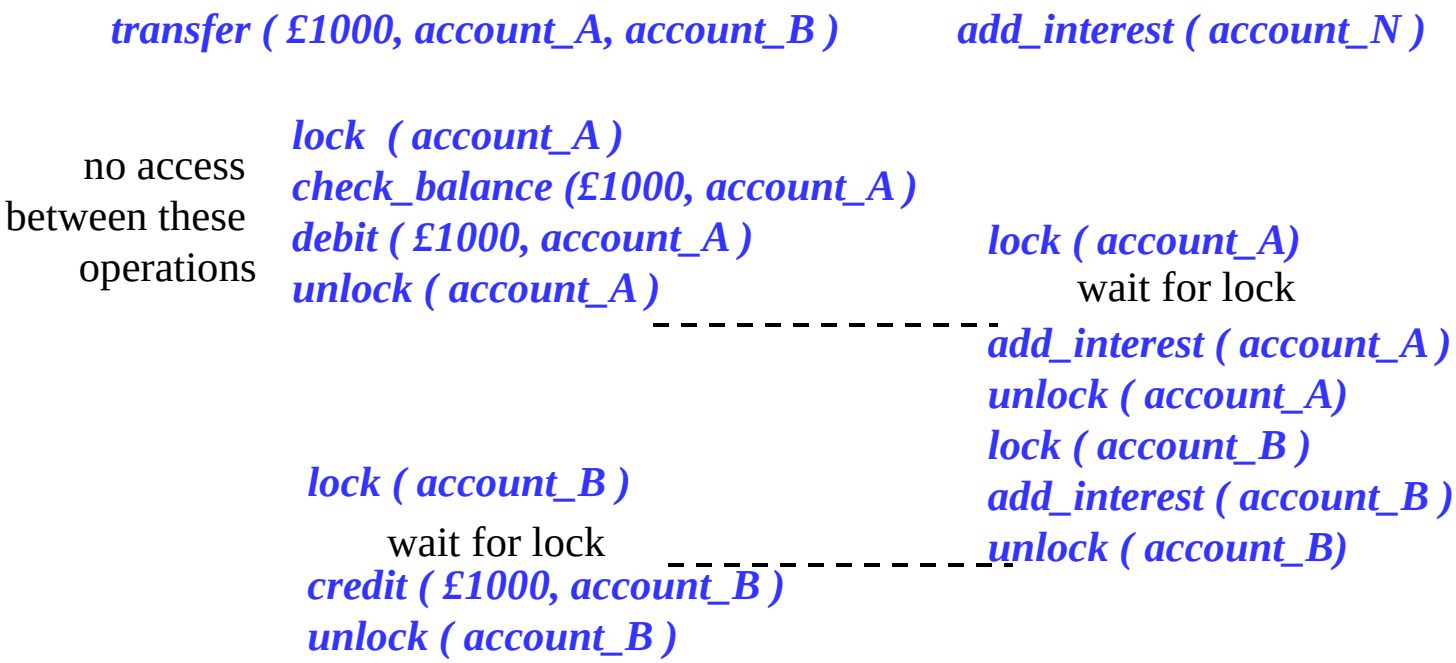
Suppose add_interest updates all accounts daily.
As before, the operations may execute correctly until an unfortunate interleaving occurs.

check_balance (£1000, account_A)	
debit (£1000, account_A)	
	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 suboperations of ***transfer***.

Object semantics – 3

Can we solve this problem by locking individual account objects before a sequence of operations on them? Add ***lock*** and ***unlock*** to the object operations:



This does not solve the problem. With unfortunate interleaving the interest on £1000 can still be lost. The database state is still (arguably) incorrect.
The effects of the suboperations of ***transfer*** are still visible.
Suppose we allow more than one object to be locked ....

Object semantics – 4

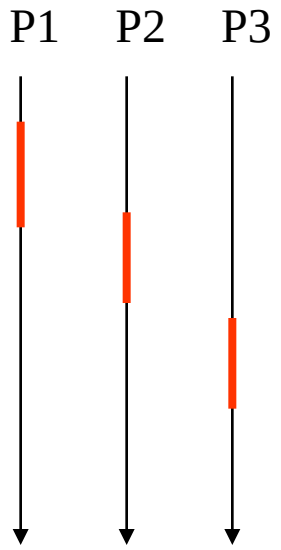
…. suppose we allow *more than one object to be locked* – e.g. for a transfer operation.

	process P	process Q
	transfer (£1000, account_A, account_B)	***add_interest (account_N)*** for all accounts
	lock (account_A)	
	check_balance (£1000, account_A)	
lock A held	***debit (£1000, account_A)***	***lock (account_A)***
lock B requested	***lock (account_B)***	wait for lock
	credit (£1000, account_B)	
	unlock (account_B)	
	unlock (account_A)	
		add_interest (account_A)
		unlock (account_A)
		lock (account_B)
		add_interest (account_B)
		unlock (account_B)

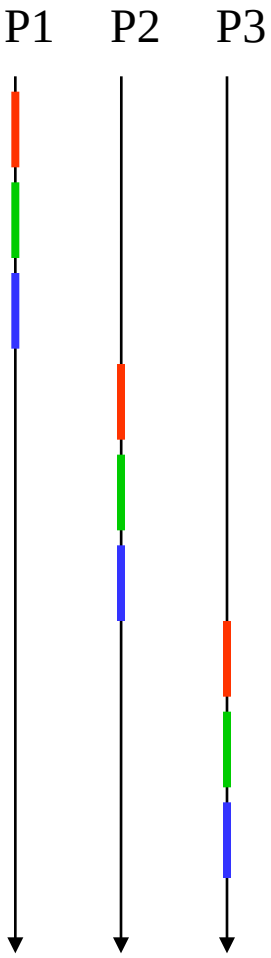
This appears to solve the problem for ***transfer*** and ***add_interest***. *
This is because ***add_interest*** does not hold locks while acquiring new locks.
Deadlock could occur with concurrent transfers, or other composite operations.
* But it doesn't! See 36.

Serialisation of composite operations - visualisation

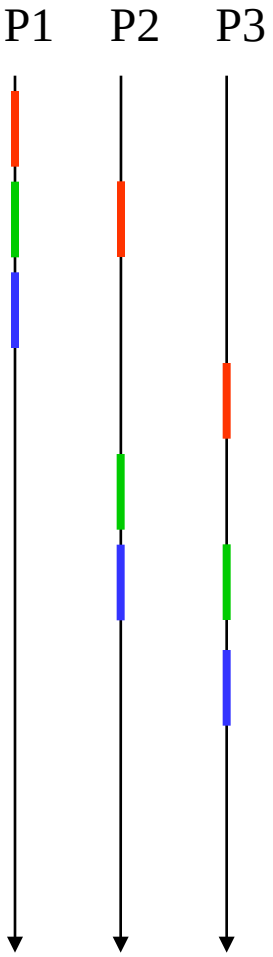
single-object/operation serialisation



composite operation strict serialisation



composite operation with interleavings – are any correct?



Transactions – notation

Transaction identifiers, commit and abort – example:

T_i = ***start$_i$,***
checkbalance$_i$ (account_A),
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 a fail, the last operation is ***abort$_i$***
On ***abort*** any intermediate effects of the transaction must be UNDONE
e.g. suppose a crash occurs after ***debit***.
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 ( £1000, account_A )
        then  transfer ( £1000, account_A, account_B); commit
        else  abort;
```

Serialisability - definition

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

Example:

serial execution:

debit (£1000, account_A)
credit (£1000, account_B)

add_interest (account_A)
add_interest (account_B)

serial execution:

add_interest (account_A)
add_interest (account_B)

debit (£1000, account_A)
credit (£1000, account_B)

non-serialisable execution

debit (£1000, account_A)

add_interest (account_A)
add_interest (account_B)

credit (£1000, account_B)

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

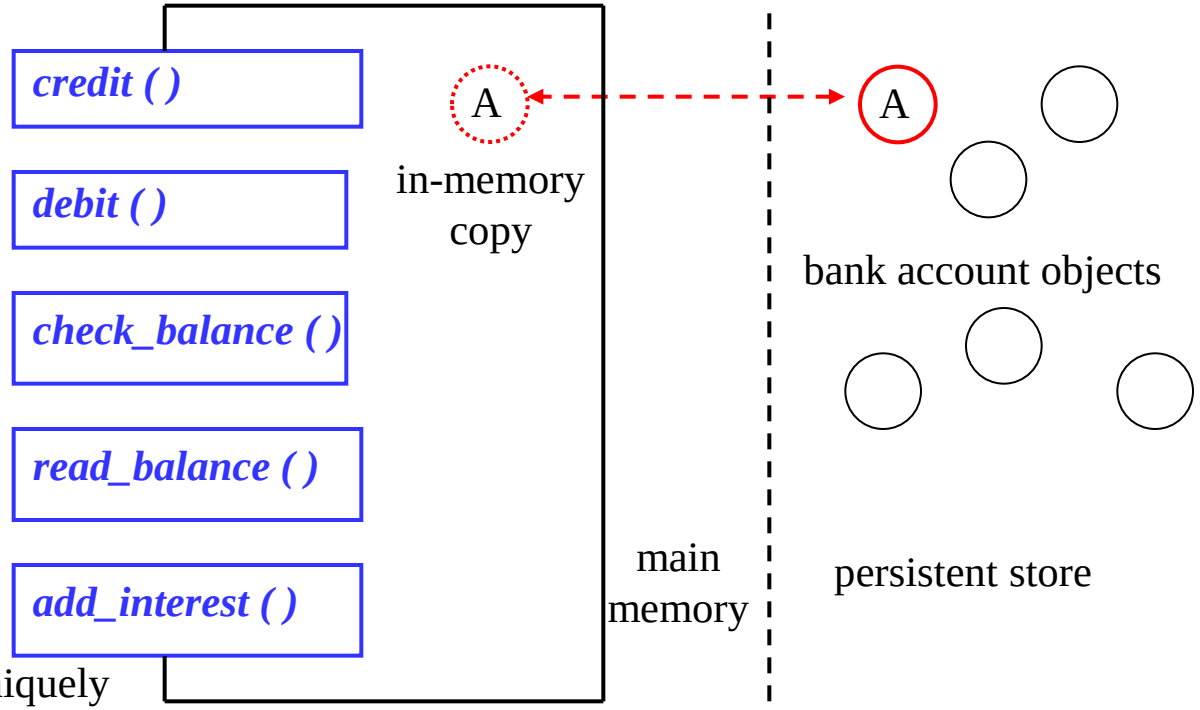
Durability the effects of a committed transaction endure/persist

C and **I** are defined with concurrency control primarily in mind,
A and **D** with requirements for crash recovery primarily in mind
But we have seen already that the mechanisms for enforcing concurrency control and crash recovery are inter-related.

Strict enforcement of **I** reduces concurrency, sometimes unnecessarily.
We investigate in slides 32 onwards whether **I** can 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



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

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

debit or ***credit*** and ***add_interest*** conflict,
credit and ***credit*** or ***debit*** and ***debit*** or ***credit*** and ***debit*** do not conflict
Arithmetic + and – do not conflict, * conflicts with + and –

Serialisability – property for implementation

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)

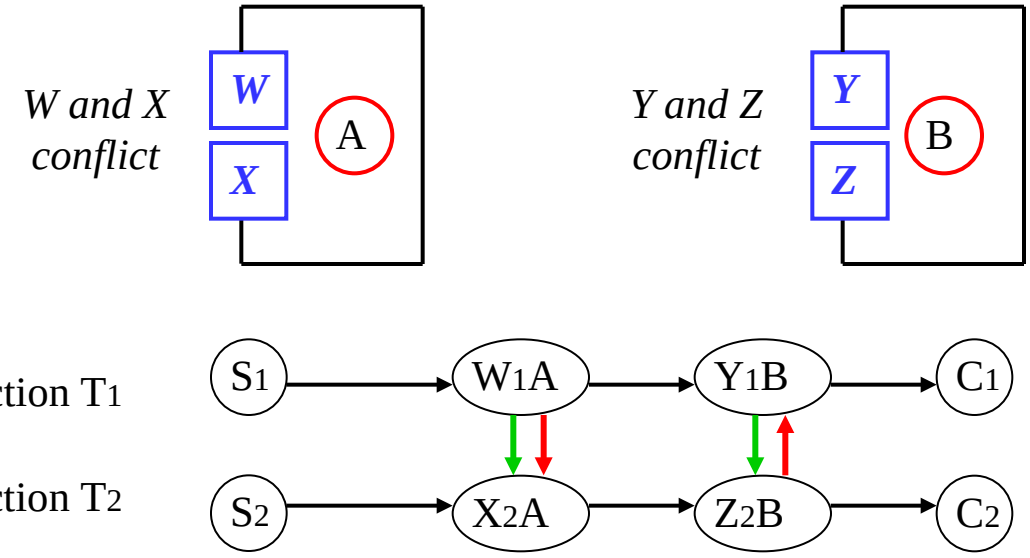
objects ***account_A*** and ***account_B*** are invoked by T1 and T2
operation ***add_interest*** conflicts with operations ***debit*** and ***credit***

object ***account_A*** T1 before T2

object ***account_B*** T2 before T1

The above operation interleavings do not form a serialisable execution

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.

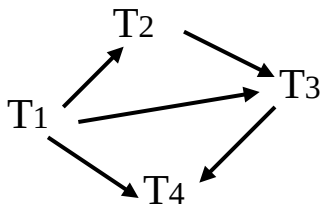
Serialisation graphs

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)

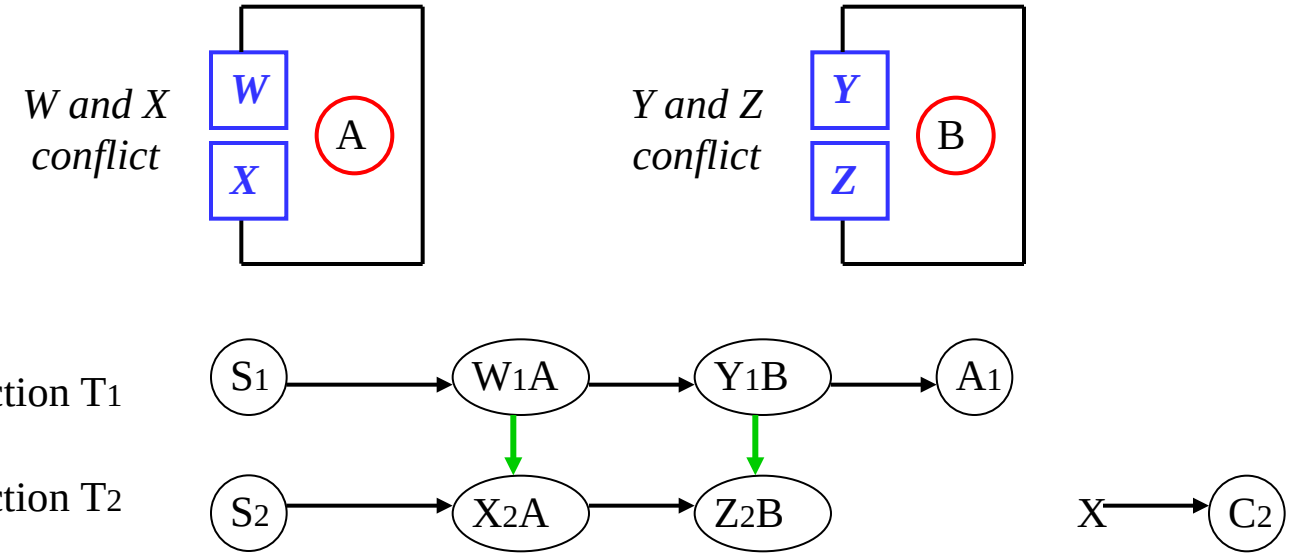
DEFINITION: A serialisable history represents a serialisable execution

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

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.
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 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
	undo3	£8000
	undo1	£7000
	redo3	£7007
	redo4	£8007
abort2	undo4	£7007
	undo3	£7000
	undo2	£5000
	redo3	£5005
	redo4	£6005
commit3		
commit4		

Computer Laboratory Technical Reports.
See http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-table.html

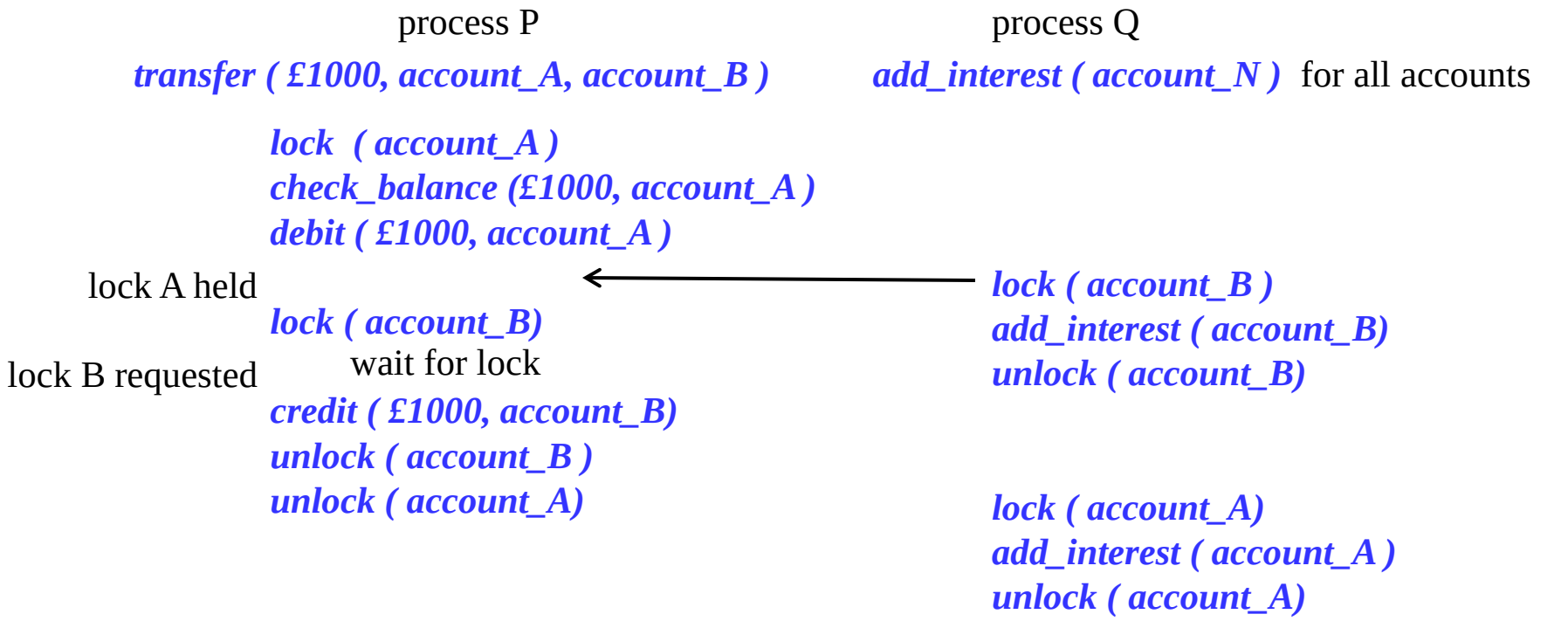
459 An open parallel architecture for data-intensive applications
Mohamad Afshar July 1999, PhD, 225p

338 A new approach to implementing atomic data types
Zhixue Wu May 1994, PhD, 170p

Reference for correctness of two-phase locking (pp. 486 – 488):

Database System Implementation
Hector Garcia-Molina, Jeffrey Ullman, Jennifer Widom
Prentice-Hall, 2000

Object semantics – 4 (from slide 23)



So-called two-phase locking, 2PL, (as above) does not solve this problem – see the above interleaving

lock (account_A) is OK - Although Process Q could get in between the two ***lock***s,
lock (account_B) Process P doesn't start changing state until it has both locks.

lock (<list of locks>) implies a lock server that interacts with all the objects.

But should Process Q lock every bank account in the system?
It's a special-case example – perhaps it's OK to make the service unavailable while interest is added?