Distributed Systems
Transactions - I

Wouter Joosen

DistriNet, KULeuven

October 31, 2023






Context of book chapters

- Shared data
  - Ch 16 Transactions and concurrency control, 16.1-16.4; 16.7
  - Ch 17 Distributed transactions
  - Ch 18 Replication

Overview

- Transactions
- Nested transactions
- Locks

Known material!

- Distributed transactions (Part 2)

- Replication (Part 3)

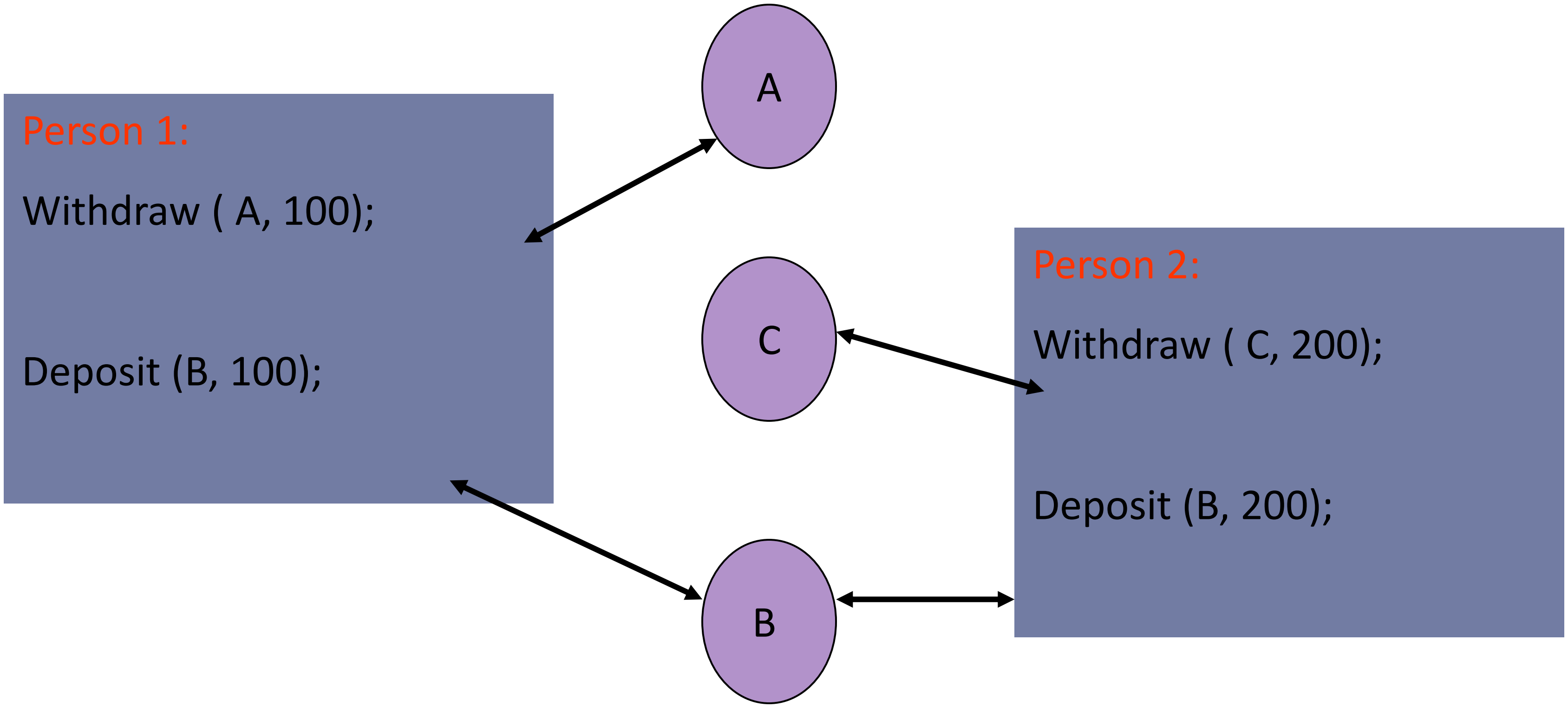
Transactions: Introduction

- Environment
  - data partitioned over different servers on different systems
  - sequence of operations as individual unit
  - long-lived data at servers (cfr. Databases)

- transactions = approach to achieve consistency of data in a distributed environment

Transactions: Introduction

- Example



Transactions: Introduction

- Critical section
  - group of instructions $\rightarrow$ indivisible block wrt other cs
  - short duration
- atomic operation (within a server)
  - operation is free of interference from operations being performed on behalf of other (concurrent) clients
  - concurrency in server $\rightarrow$ multiple threads
  - atomic operation <> critical section
- *transaction*

Transactions: Introduction

- *Critical section*
- *atomic operation*
- transaction
  - group of different operations + properties
  - single transaction may contain operations on different servers
  - possibly long duration

ACID properties

Transactions: ACID

- Properties concerning the sequence of operations that read or modify shared data:

  A tomicity

  C onsistency

  I solation

  D urability

Transactions: ACID

- Atomicity or the "all-or-nothing" property
  - a transaction
    - commits = completes successfully or
    - aborts = has no effect at all
  - the effect of a committed transaction
    - is guaranteed to persist
    - can be made visible to other transactions
  - transaction aborts can be initiated by
    - the system (e.g. when a node fails) or
    - a user issuing an abort command

Transactions: ACID

- Consistency
  - a transaction moves data from one consistent state to another
- Isolation
  - no interference from other transactions
  - intermediate effects invisible to other transactions

  The isolation property has 2 parts:
  - serializability: running concurrent transactions has the same effect as some serial ordering of the transactions
  - Failure isolation: a transaction cannot see the uncommitted effects of another transaction

Transactions: ACID

- Durability
  - once a transaction commits, the effects of the transaction are preserved despite subsequent failures

Transactions: Life histories

- Transactional service operations
  - *OpenTransaction()* $\rightarrow$ Trans
    - starts new transaction
    - returns unique identifier for transaction
  - *CloseTransaction(Trans)* $\rightarrow$ (Commit, Abort)
    - ends transaction
    - returns commit if transaction committed else abort
  - *AbortTransaction(Trans)*
    - aborts transaction

Transactions: Life histories

- History 1: success

```
T := OpenTransaction();
            operation;
            operation;
            ….
            operation;
CloseTransaction(T);
```

Operations have
read
or
write
semantics

Transactions: Life histories

- History 2: abort by client

```
T := OpenTransaction();
        operation;
        operation;
        ….
        operation;
AbortTransaction(T);
```

Transactions: Life histories

- History 3: abort by server

```
T := OpenTransaction();
        operation;
        operation;
        ….
        operation;
```

Server aborts!

Error reported

Transactions: Concurrency

- Illustration of well known problems:
  - the lost update problem
  - inconsistent retrievals
- operations used + implementations
  - Withdraw(A, n)

```
b := A.read();
A.write( b - n);
```

  - Deposit(A, n)

```
b := A.read();
A.write( b + n);
```

REPEATS KNOWN MATERIAL

Transactions: Concurrency

- The lost update problem:

Transaction T	Transaction U
Withdraw(A,4);	Withdraw(C,3);
Deposit(B,4);	Deposit(B,3);

Interleaved execution of operations on B → ?

Transactions: Concurrency

- The lost update problem:

Transaction T A ➔ B: 4	Transaction U C ➔ B: 3

```
bt := A.read();

A.write(bt-4);
```

A: 100

B: 200

C: 300

Transactions: Concurrency

- The lost update problem:

Transaction T A ➔ B: 4		Transaction U C ➔ B: 3
bt := A.read();	A: 96	
A.write(bt-4);		bu := C.read();
	B: 200	C.write(bu-3);
	C: 300	

Transactions: Concurrency

- The lost update problem:

Transaction T A ➔ B: 4		Transaction U C ➔ B: 3
bt := A.read();	A: 96	
A.write(bt-4);		bu := C.read();
	B: 200	C.write(bu-3);
bt := B.read();		
bt=200	C: 297	bu := B.read();
		B.write(bu+3);

Transactions: Concurrency

- The lost update problem:

Transaction T A → B: 4		Transaction U C → B: 3
bt := A.read();		
A.write(bt-4);	A: 96	
		bu := C.read();
	B: 203	C.write(bu-3);
bt := B.read();		
bt=200	C: 297	bu := B.read();
		B.write(bu+3);
B.write(bt+4);		

Transactions: Concurrency

- The lost update problem:

Transaction T A → B: 4		Transaction U C → B: 3
bt := A.read();		
A.write(bt-4);	A: 96	
		bu := C.read();
	B: 204	C.write(bu-3);
bt := B.read();		
bt=200	C: 297	bu := B.read();
		B.write(bu+3);
B.write(bt+4);		

Correct B = 207!!

Transactions: Concurrency

- The inconsistent retrieval problem:

Transaction T	Transaction U
Withdraw(A,50);	BranchTotal();
Deposit(B,50);	

Transactions: Concurrency

- The inconsistent retrieval problem :

Transaction T A → B: 50	Transaction U BranchTotal
bt := A.read();	
A.write(bt-50);	

A: 100

B: 200

C: 300

Transactions: Concurrency

- The inconsistent retrieval problem :

Transaction T A → B: 50		Transaction U BranchTotal
bt := A.read();	A: 50	
A.write(bt-50);		bu := A.read();
	B: 200	bu := bu + B. read();
	C: 300	bu := bu + C.read();
bt := B.read();		550
B.write(bt+50);		

Transactions: Concurrency

- The inconsistent retrieval problem:

Transaction T A ➔ B: 50		Transaction U BranchTotal
bt := A.read();	A: 50	
A.write(bt-50);		bu := A.read();
	B: 250	bu := bu + B. read();
	C: 300	bu := bu + C.read();
bt := B.read();		550
B.write(bt+50);	Correct total: 600	

Transactions: Concurrency

- Illustration of well known problems:
  - the lost update problem
  - inconsistent retrievals
- elements of solution
  - execute all transactions serially?
    - No concurrency ➔ unacceptable
  - execute transactions in such a way that overall execution is equivalent with some serial execution
    - sufficient? Yes
    - how? Concurrency control

Transactions: Concurrency

- The lost update problem: serially equivalent interleaving

Transaction T A → B: 4

bt := A.read();

A.write(bt-4);

Transaction U C → B: 3

A: 100

B: 200

C: 300

Transactions: Concurrency

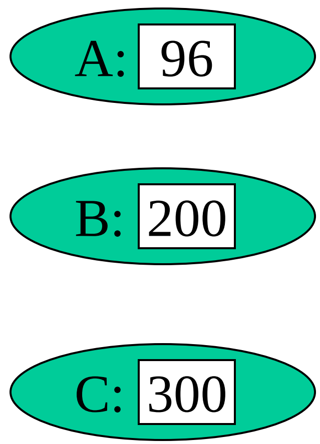
- The lost update problem: serially equivalent interleaving

Transaction T A → B: 4

Transaction U C → B: 3

bt := A.read();

A.write(bt-4);



bu := C.read();

C.write(bu-3);

Transactions: Concurrency

- The lost update problem: serially equivalent interleaving

Transaction T A → B: 4

bt := A.read();

A.write(bt-4);

bt := B.read();

B.write(bt+4);

A: 96

B: 200

C: 297

Transaction U C → B: 3

bu := C.read();

C.write(bu-3);

Transactions: Concurrency

- The lost update problem: serially equivalent interleaving

Transaction T A → B: 4		Transaction U C → B: 3
bt := A.read();		
	A: 96	
A.write(bt-4);		
		bu := C.read();
	B: 204	C.write(bu-3);
bt := B.read();		
B.write(bt+4);	C: 297	bu := B.read();
		B.write(bu+3);

Transactions: Concurrency

- The lost update problem: serially equivalent interleaving

Transaction T A → B: 4

```
bt := A.read();
A.write(bt-4);
bt := B.read();
B.write(bt+4);
```

A: 96

B: 207

C: 297

Transaction U C → B: 3

```
bu := C.read();
C.write(bu-3);
bu := B.read();
B.write(bu+3);
```

Transactions: Recovery

- Illustration of well known problems:
  - a dirty read
  - premature write
- operations used + implementations

REPEATS

KNOWN

MATERIAL

  - Withdraw(A, n)

```
b := A.read();
A.write( b - n);
```

  - Deposit(A, n)

```
b := A.read();
A.write( b + n);
```

Transactions: Recovery

- A dirty read problem:

Transaction T	Transaction U
Deposit(A,4);	Deposit(A,3);

Interleaved execution and abort ➔ ?

Transactions: Recovery

- A dirty read problem:

Transaction T 4 ➔ A		Transaction U 3➔ A
bt := A.read();	A: $\boxed{100}$	
A.write(bt+4);		

Transactions: Recovery

- A dirty read problem:

Transaction T 4 ➔ A		Transaction U 3➔ A
bt := A.read();		
	A: 104	
A.write(bt+4);		
		bu := A.read();
		A.write(bu+3);

Transactions: Recovery

- A dirty read problem:

Transaction T 4 ➔ A		Transaction U 3➔ A
bt := A.read();	A: 107	
A.write(bt+4);		
		bu := A.read();
		A.write(bu+3);
		Commit
Abort		

Correct result: A = 103

Transactions: Recovery

- Premature write or
  Over-writing uncommitted values :

Transaction T	Transaction U
Deposit(A,4);	Deposit(A,3);

Interleaved execution and Abort ➔ ?

Transactions: Recovery

- Over-writing uncommitted values :

Transaction T 4 → A

bt := A.read();

A.write(bt+4);

A: $\boxed{100}$

Transaction U 3→ A

Transactions: Recovery

- Over-writing uncommitted values :

Transaction T 4 ➔ A		Transaction U 3➔ A
bt := A.read();		
A.write(bt+4);	A: 104	
		bu := A.read();
		A.write(bu+3);

Transactions: Recovery

- Over-writing uncommitted values :

Transaction T 4 ➔ A		Transaction U 3➔ A
bt := A.read();		
A.write(bt+4);	A: 107	
		bu := A.read();
		A.write(bu+3);
		Abort

Correct result: A = 104

Transactions: Recovery

- Illustration of well known problems:
  - a dirty read
  - premature write
- elements of solution:
  - **Cascading Aborts**: a transaction reading uncommitted data must be aborted if the transaction that modified the data aborts
  - to avoid cascading aborts, transactions can only read data written by committed transactions
  - undo of write operations must be possible

…Transactions: Recovery (!)

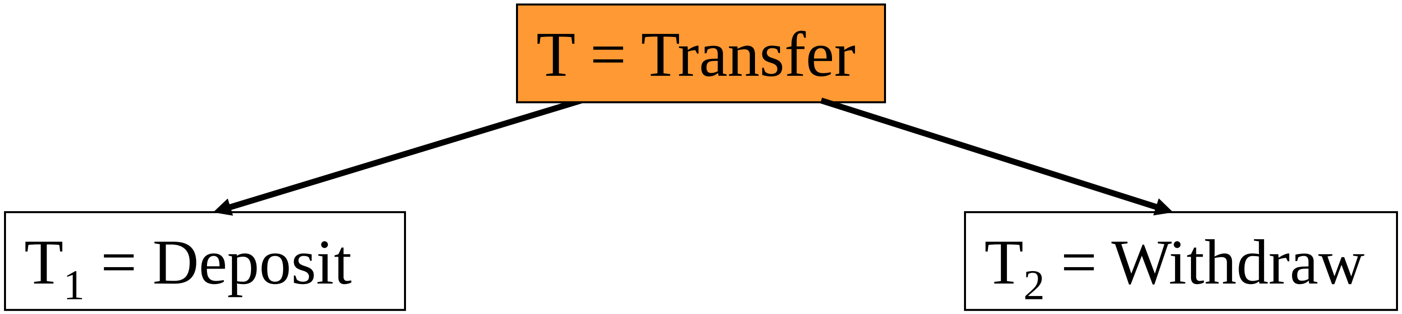
- how to preserve data despite subsequent failures?
  - usually by using **stable** storage
    - ***two*** copies of data stored
      - in separate parts of disks
      - not decay related (probability of both parts corrupted is small)

Nested Transactions

- Transactions composed of several sub-transactions

- Why nesting?
  - Modular approach to structuring transactions in applications
  - means of controlling concurrency within a transaction
    - concurrent sub-transactions accessing shared data are serialized
  - a finer grained recovery from failures
    - sub-transactions fail independent

Nested Transactions



- Sub-transactions commit or abort independently
  - without effect on outcome of other sub-transactions or enclosing transactions
- effect of sub-transaction becomes durable only when top-level transaction commits

Concurrency control: locking

- Environment
  - shared data in a single server (this section)
  - many competing clients
- problem:
  - realize transactions
  - maximize concurrency
- solution: serial equivalence

Concurrency control: locking

- Protocols:
  - Locks
  - Optimistic Concurrency Control
  - Timestamp Ordering

Concurrency control: locking

- Protocols:
  - Locks
  - Optimistic Concurrency Control
  - Timestamp Ordering

Concurrency control: locking

- Example:
  - access to shared data within a transaction
    ➔ lock (= data reserved for …)
  - exclusive locks
    - exclude access by other transactions

Concurrency control: locking

- Same example (lost update) with locki

REPEATS

KNOWN

MATERIAL

Transaction T	Transaction U
Withdraw(A,4);	Withdraw(C,3);
Deposit(B,4);	Deposit(B,3);

Colour of data show owner of lock

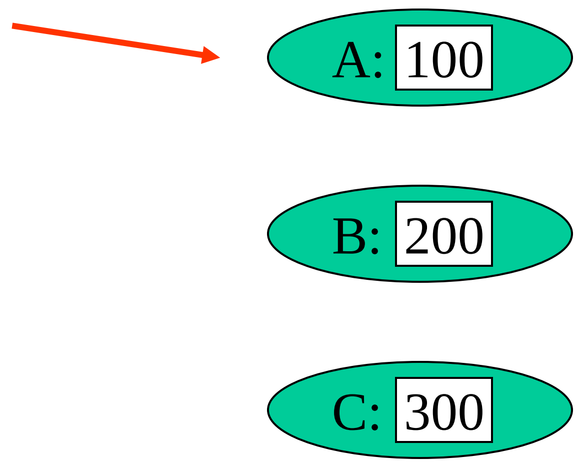
Concurrency control: locking

- Exclusive locks

Transaction T A → B: 4

Transaction U C → B: 3

bt := A.read();



Concurrency control: locking

- Exclusive locks

Transaction T A → B: 4

Transaction U C → B: 3

bt := A.read();

A.write(bt-4);

A: 100

B: 200

C: 300

Concurrency control: locking

- Exclusive locks

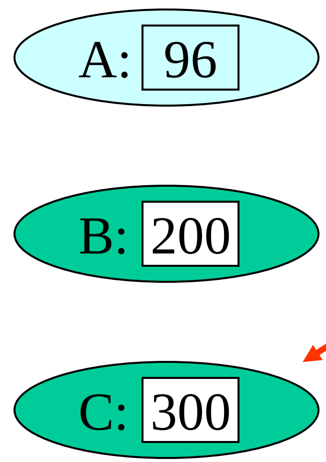
Transaction T A ➔ B: 4

bt := A.read();

A.write(bt-4);

Transaction U C ➔ B: 3

bu := C.read();



Concurrency control: locking

- Exclusive locks

Transaction T A → B: 4

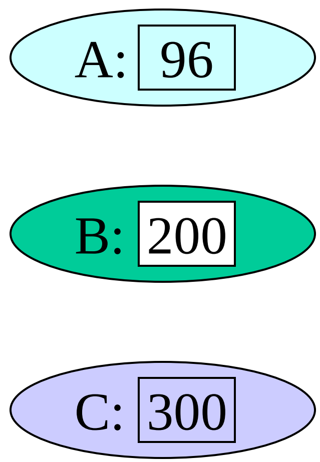
bt := A.read();

A.write(bt-4);

Transaction U C → B: 3

bu := C.read();

C.write(bu-3);



Concurrency control: locking

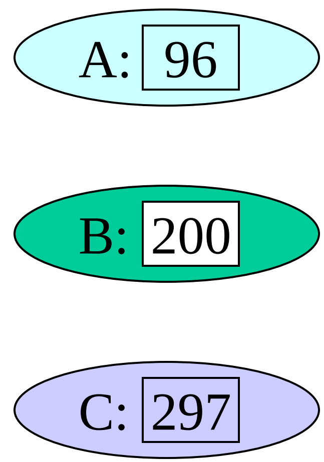
- Exclusive locks

Transaction T A ➔ B: 4

bt := A.read();

A.write(bt-4);

bu := B.read();



Transaction U C ➔ B: 3

bu := C.read();

C.write(bu-3);

Concurrency control: locking

- Exclusive locks

Transaction T A → B: 4		Transaction U C → B: 3
bt := A.read();	A: 96	
A.write(bt-4);		bu := C.read();
	B: 200	C.write(bu-3);
bt := B.read();		Wait for T
		bu := B.read();
	C: 297	
B.write(bt+4);		

Concurrency control: locking

- Exclusive locks

Transaction T A ➔ B: 4

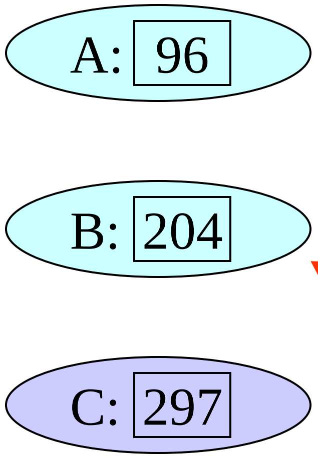
bt := A.read();

A.write(bt-4);

bt := B.read();

B.write(bt+4);

CloseTransaction(T);



Transaction U C ➔ B: 3

bu := C.read();

C.write(bu-3);

Wait for T

bu := B.read();

Concurrency control: locking

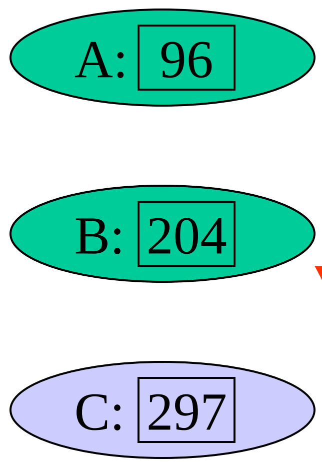
- Exclusive locks

Transaction T A ➔ B: 4

Transaction U C ➔ B: 3

bt := A.read();

A.write(bt-4);



bu := C.read();

C.write(bu-3);

bt := B.read();

Wait for T

bu := B.read();

B.write(bt+4);

CloseTransaction(T);

Concurrency control: locking

- Exclusive locks

Transaction T A → B: 4

bt := A.read();

A.write(bt-4);

bt := B.read();

B.write(bt+4);

CloseTransaction(T);

A: 96

B: 204

C: 297

Transaction U C → B: 3

bu := C.read();

C.write(bu-3);

bu := B.read();

B.write(bu+3);

Concurrency control: locking

- Exclusive locks

Transaction T A ➔ B: 4		Transaction U C ➔ B: 3
bt := A.read();		
A.write(bt-4);	A: 96	
		bu := C.read();
	B: 207	C.write(bu-3);
bt := B.read();		
	C: 297	bu := B.read();
B.write(bt+4);		
CloseTransaction(T);		
		B.write(bu+3);
		CloseTransaction(U);

Concurrency control: locking

- Exclusive locks

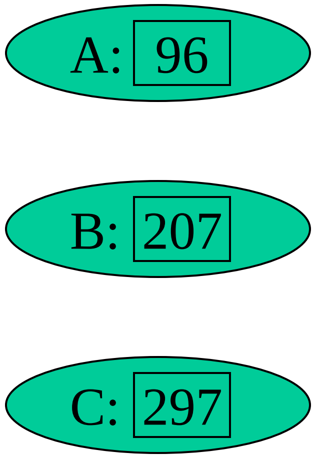
Transaction T A → B: 4

```
bt := A.read();
A.write(bt-4);

bt := B.read();

B.write(bt+4);

CloseTransaction(T);
```



Transaction U C → B: 3

```
bu := C.read();
C.write(bu-3);

bu := B.read();

B.write(bu+3);
CloseTransaction(U);
```

Concurrency control: locking

- Basic elements of protocol

  1 serial equivalence

  - requirements
    - – all of a transaction's accesses to a particular data item should be serialized with respect to accesses by other transactions
    - – all pairs of conflicting operations of 2 transactions should be executed in the same order
  - how?
    - – A transaction is not allowed any new locks after it has released a lock

  ➔ Two-phase locking

Concurrency control: locking

- Two-phase locking
  - Growing Phase
    - new locks can be acquired
  - Shrinking Phase
    - no new locks
    - locks are released

Concurrency control: locking

- Basic elements of protocol

  1 *serial equivalence* ➔ *two-phase locking*

  2 hide intermediate results

  - conflict between
    - release of lock access by other transactions possible
    - access should be delayed till commit/abort transaction
  - how?
    - New mechanism?
    - (better) release of locks only at commit/abort

  ➔ strict two-phase locking
    - locks held till end of transaction

Concurrency control: locking

- How increase concurrency and preserve serial equivalence?
  - Granularity of locks
  - Appropriate locking rules

Concurrency control: locking

- Granularity of locks
  - observations
    - large number of data items on server
    - typical transaction needs only a few items
    - conflicts unlikely
  - large granularity
    - ➔limits concurrent access
    - example: all accounts in a branch of bank are locked together
  - small granularity
    - ➔ overhead

Concurrency control: locking

- Appropriate locking rules
  - when conflicts?

operation by T	operation by U	conflict
read	read	No
read	write	Yes
write	write	Yes

➔ Read & Write locks

Concurrency control: locking

- Lock compatibility

For one data item		Lock requested	
		Read	Write
Lock already set	None	OK	OK
	Read	OK	Wait
	Write	Wait	Wait

Concurrency control: locking

- Strict two-phase locking
  - locking
    - done by server (containing data item)
  - unlocking
    - done by commit/abort of the transactional service

Concurrency control: locking

- Use of locks on strict two-phase locking
  - when an operation accesses a data item
    - not locked yet

      ➔ lock set & operation proceeds
    - conflicting lock set by another transaction

      ➔transaction must wait till ...
    - non-conflicting lock set by another transaction

      ➔lock shared & operation proceeds
    - locked by same transaction

      ➔lock promoted if necessary & operation proceeds

Concurrency control: locking

- Use of locks on strict two-phase locking
  - *when an operation accesses a data item*

  - when a transaction is committed/aborted
    - ➔ server unlocks all data items locked for the transaction

Concurrency control: locking

- Lock implementation
  - lock manager
  - managing table of locks:
    - transaction identifiers
    - identifier of (locked) data item
    - lock type
    - condition variable
      - for waiting transactions

Concurrency control: locking

- Deadlocks
  - a state in which each member of a group of transactions is waiting for some other member to release a lock

  ➔no progress possible!

  - Example: with read/write locks

Concurrency control: locking

- Same example (lost update) with locking

Transaction T	Transaction U
Withdraw(A,4);	Withdraw(C,3);
Deposit(B,4);	Deposit(B,3);

Colour of data show owner of lock

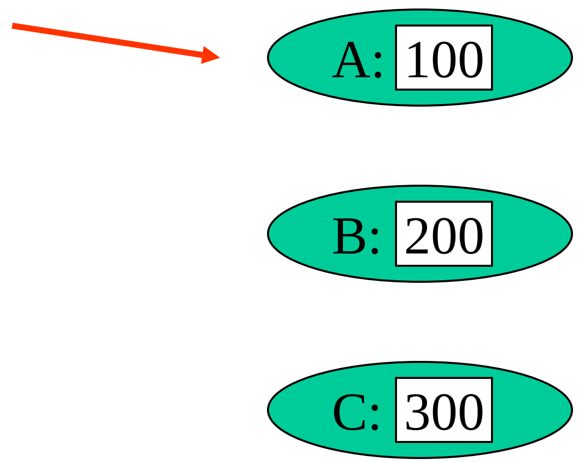
Concurrency control: locking

- Read/write locks

Transaction T A → B: 4

Transaction U C → B: 3

bt := A.read();



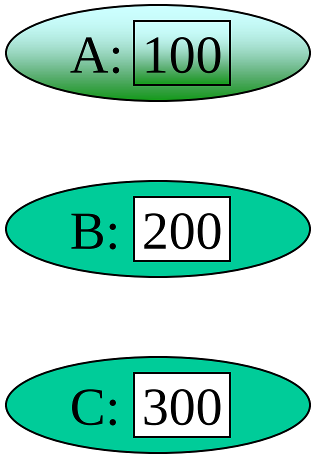
Concurrency control: locking

- Read/write locks

Transaction T A ➔ B: 4

Transaction U C ➔ B: 3

```
bt := A.read();
A.write(bt-4);
```



Concurrency control: locking

- Read/write locks

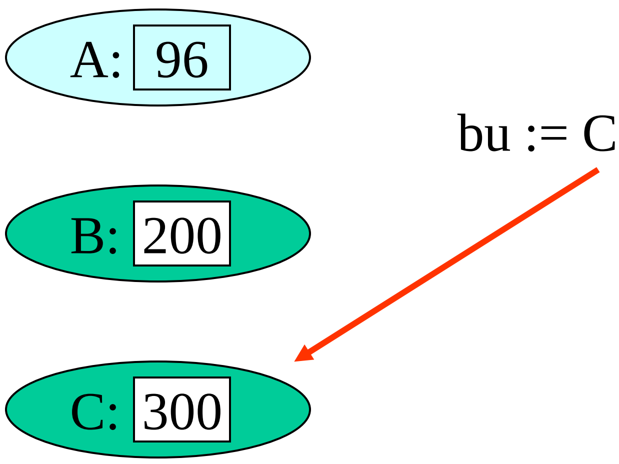
Transaction T A → B: 4

Transaction U C → B: 3

bt := A.read();

A.write(bt-4);

bu := C.read();



Concurrency control: locking

- Read/write locks

Transaction T A ➔ B: 4

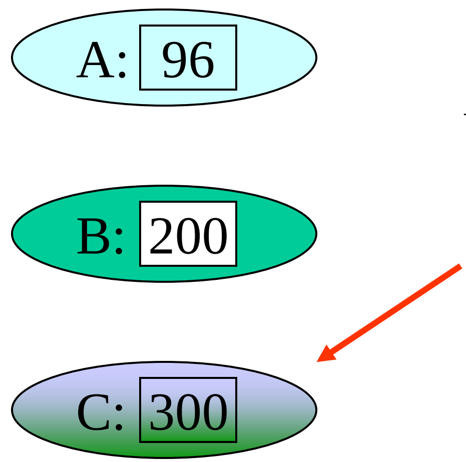
Transaction U C ➔ B: 3

bt := A.read();

A.write(bt-4);

bu := C.read();

C.write(bu-3);



Concurrency control: locking

- Read/write locks

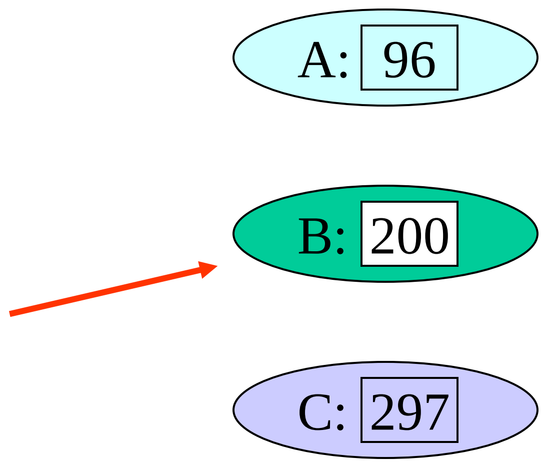
Transaction T A ➔ B: 4

Transaction U C ➔ B: 3

bt := A.read();

A.write(bt-4);

bu := B.read();



bu := C.read();

C.write(bu-3);

Concurrency control: locking

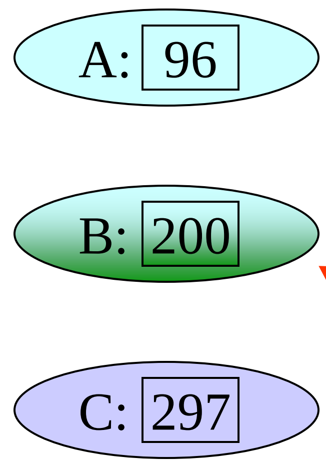
- Read/write locks

Transaction T A → B: 4

bt := A.read();

A.write(bt-4);

bt := B.read();



Transaction U C → B: 3

bu := C.read();

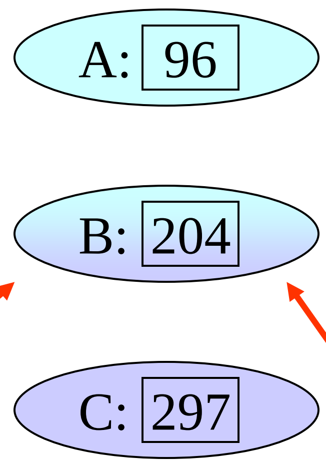
C.write(bu-3);

bu := B.read();

Concurrency control: locking

- Read/write lock

Transaction T A ➔ B: 4	Transaction U C ➔ B: 3
bt := A.read();	
A.write(bt-4);	
	bu := C.read();
	C.write(bu-3);
bt := B.read();	
	bu := B.read();
Wait for release by U	
B.write(bt+4);	
	Wait for release by T
	B.write(bu+3);



Deadlock!!

Concurrency control: locking

- Solutions to the Deadlock problem
  - Prevention
    - by locking all data items used by a transaction when it starts
    - by requesting locks on data items in a predefined order

    Evaluation
    - impossible for interactive transactions
    - reduction of concurrency

Concurrency control: locking

- Solutions to the Deadlock problem
  - Detection
    - the server keeps track of a wait-for graph
      - lock: edge is added
      - unlock: edge is removed
    - the presence of cycles may be checked
      - when an edge is added
      - periodically
  - example

Concurrency control: locking

- Read/write locks

Transaction T A ➔ B: 4

bt := A.read();

A.write(bt-4);

bt := B.read();

A: 96

B: 200

C: 297

Transaction U C ➔ B: 3

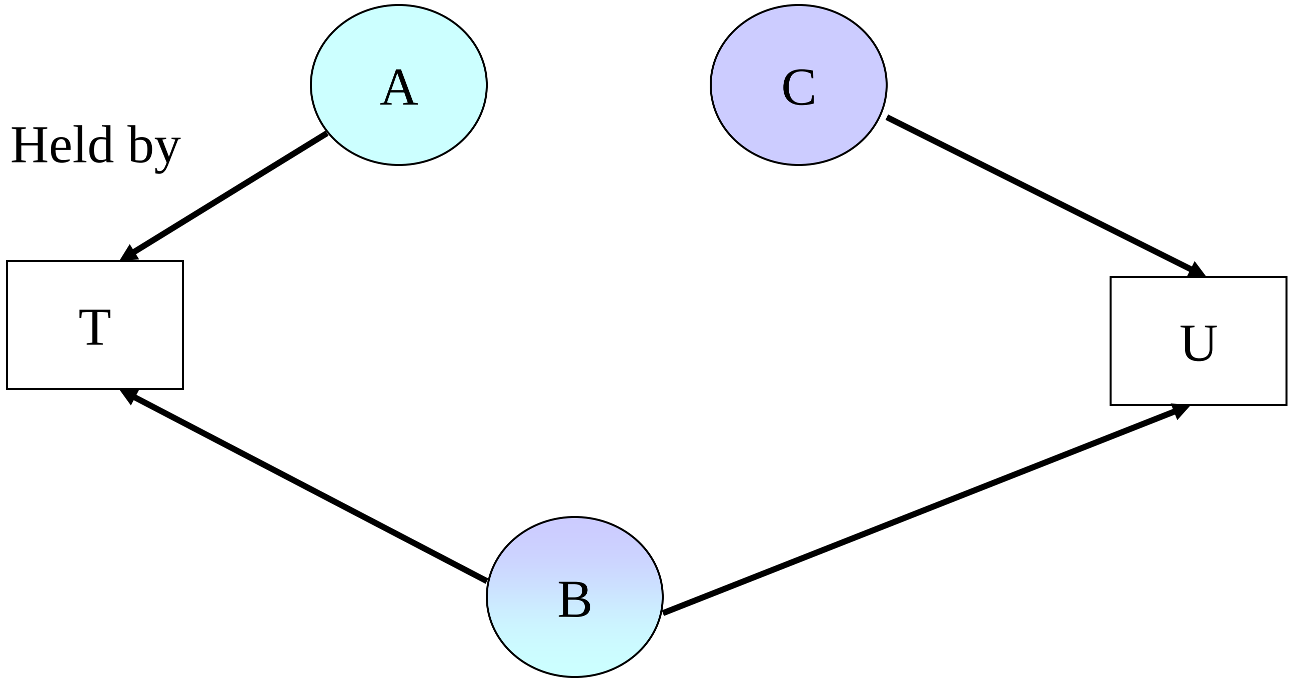
bu := C.read();

C.write(bu-3);

bu := B.read();

Concurrency control: locking

- Wait-for graph



Concurrency control: locking

- Read/write locks

Transaction T A ➔ B: 4

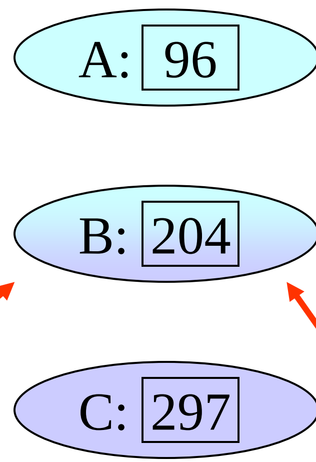
bt := A.read();

A.write(bt-4);

bt := B.read();

Wait for release by U

B.write(bt+4);



Transaction U C ➔ B: 3

bu := C.read();

C.write(bu-3);

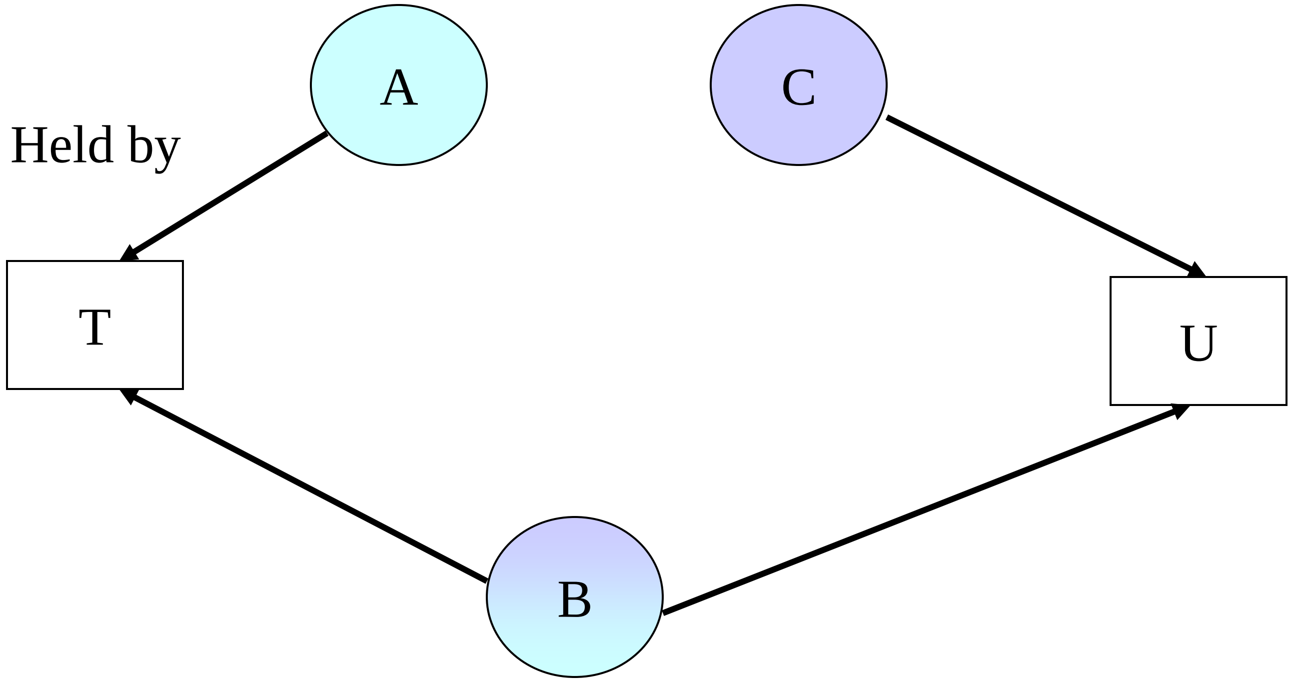
bu := B.read();

Wait for release by T

B.write(bu+3);

Concurrency control: locking

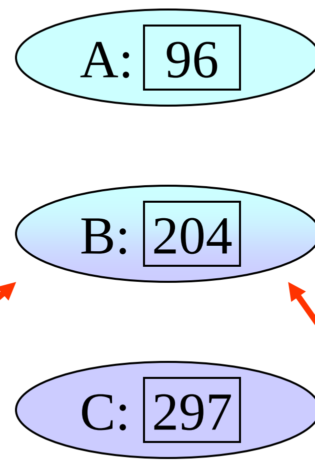
- Wait-for graph



Concurrency control: locking

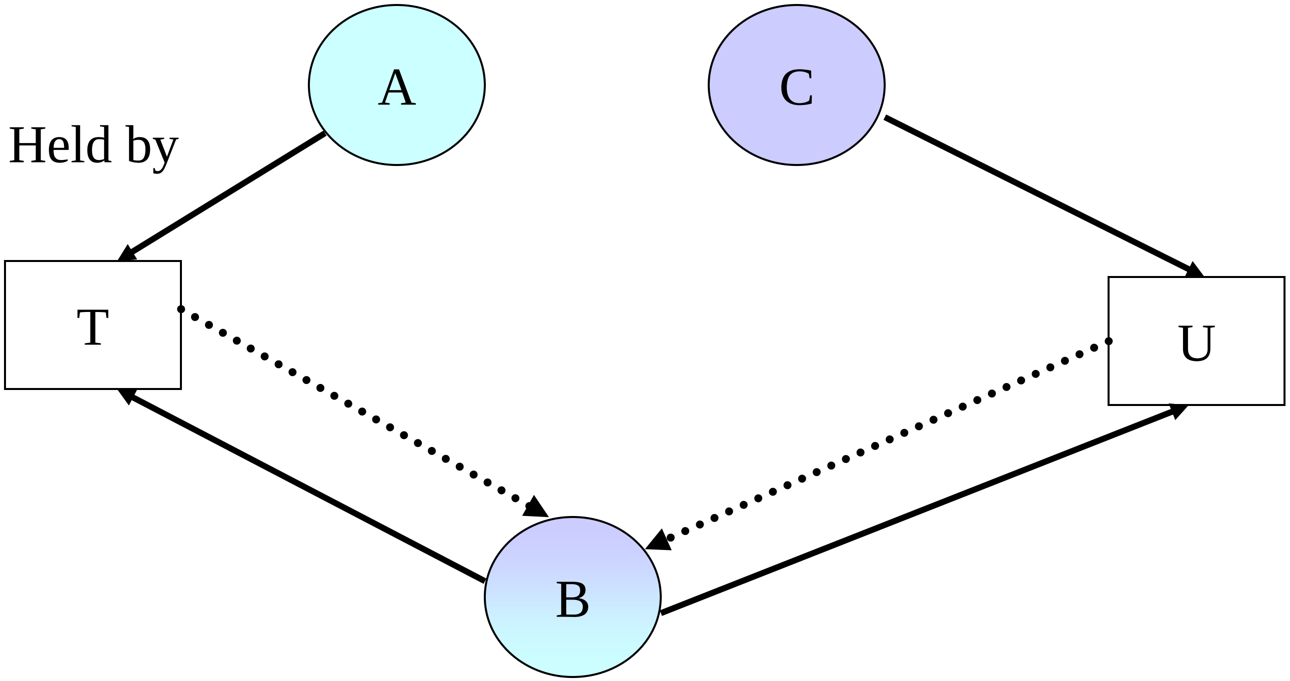
- Read/write locks

Transaction T A ➔ B: 4	Transaction U C ➔ B: 3
bt := A.read();	
A.write(bt-4);	
	bu := C.read();
	C.write(bu-3);
bt := B.read();	
	bu := B.read();
Wait for release by U	
	Wait for release by T
B.write(bt+4);	
	B.write(bu+3);



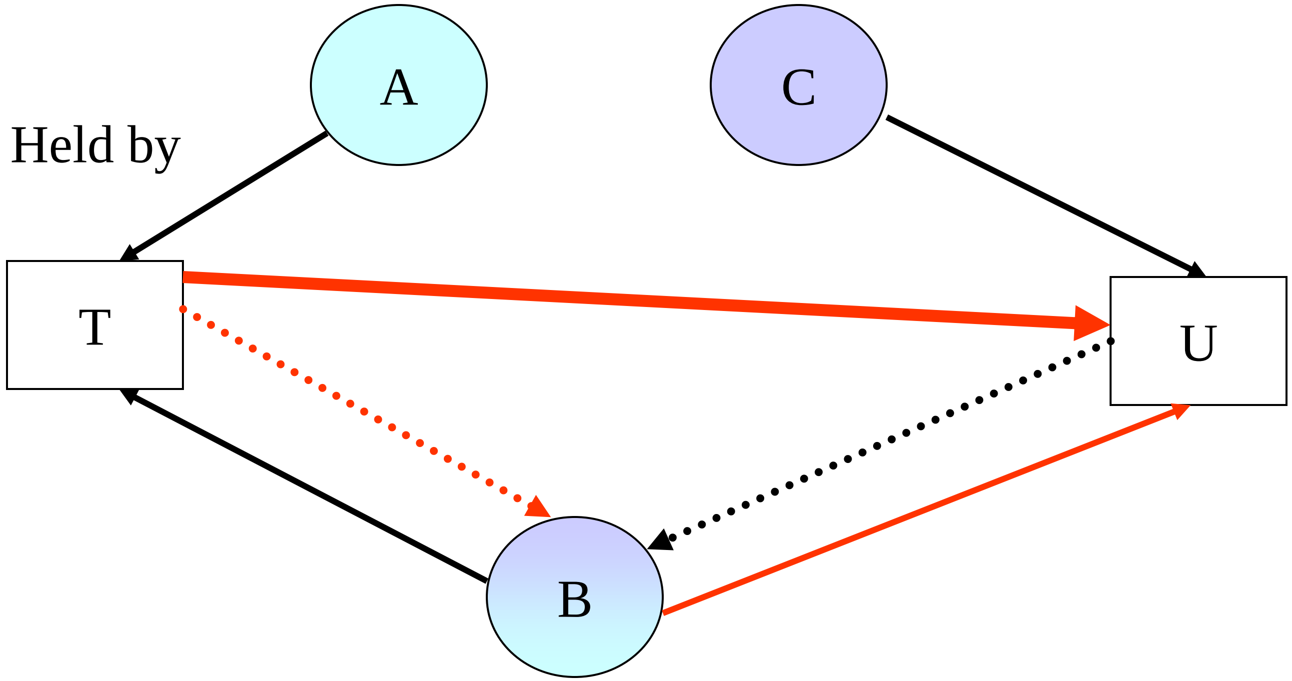
Concurrency control: locking

- Wait-for graph



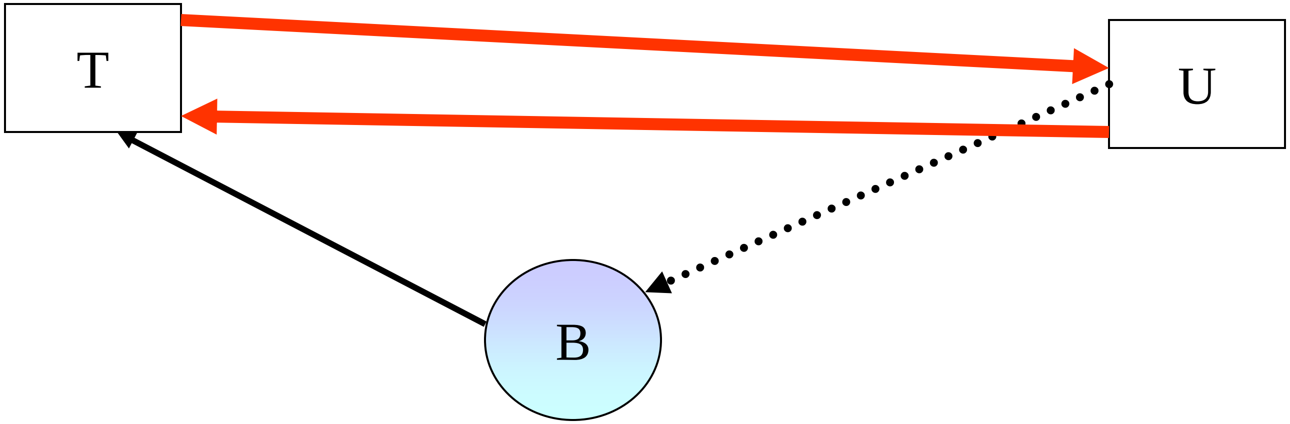
Concurrency control: locking

- Wait-for graph



Concurrency control: locking

- Wait-for graph



Cycle ➔ deadlock

Concurrency control: locking

- Solutions to the Deadlock problem
  - Detection
    - the server keeps track of a wait-for graph
    - the presence of cycles must be checked
    - once a deadlock detected, the server must select a transaction and abort it (to break the cycle)
    - choice of transaction? Important factors
      - age of transaction
      - number of cycles the transaction is involved in

Concurrency control: locking

- Solutions to the Deadlock problem
  - Timeouts
    - locks granted for a limited period of time
      - within period: lock invulnerable
      - after period: lock vulnerable

Overview

- Part 1: Transactions - recap

- Part 2: *Distributed* transactions
  - Flat and nested distributed transactions
  - Atomic commit protocols
  - Concurrency in distributed transactions
  - Distributed deadlocks
  - Transaction recovery