Concurrent systems

Lecture 5: Concurrency without shared data, composite operations and transactions, and serialisability

Dr Robert N. M. Watson

1

Reminder from last time

- Liveness properties
- Deadlock (requirements; resource allocation graphs; detection; prevention; recovery)

Concurrency is **so** hard!

If only there were some way that programmers could accomplish useful concurrent computation without…

(1) the hassles of shared memory concurrency
(2) blocking synchronisation primitives

This time

- Concurrency without shared data
  - Active objects
- **Message passing**; the actor model
  - Linda, occam, Erlang
- **Composite operations**
  - **Transactions**, **ACID properties**
  - **Isolation** and **serialisability**
- **History graphs**; **good** (and **bad**) **schedules**

This material has significant overlap with **databases** and **distributed systems** – but is presented here from a concurrency perspective

Concurrency without shared data

- The examples so far have involved threads which can arbitrarily read & write shared data
  - A key need for mutual exclusion has been to avoid race-conditions (i.e. 'collisions' on access to this data)
- An alternative approach is to have only one thread access any particular piece of data
  - Different threads can own distinct chunks of data
- Retain concurrency by allowing other threads to ask for operations to be done on their behalf
  - This 'asking' of course needs to be concurrency safe…

Fundamental design dimension: concurrent access via **shared data** vs. concurrent access via **explicit communication**

Example: Active Objects

- A monitor with an associated **server** thread
  - Exports an **entry** for each operation it provides
  - Other (**client**) threads 'call' methods
  - Call returns when operation is done
- All complexity bundled up in an **active object**
  - Must manage mutual exclusion where needed
  - Must queue requests from multiple threads
  - May need to delay requests pending conditions
    - E.g. if a producer wants to insert but buffer is full

Observation: code running in **exactly** one thread, and the data that only it accesses, effectively experience **mutual exclusion**

Producer-Consumer in Ada

```
task-body ProducerConsumer is
  ...
  loop
    SELECT
      when count < buffer-size
        ACCEPT insert(item) do
          // insert item into buffer
        end;
      count++;
    or
      when count > 0
        ACCEPT consume(item) do
          // remove item from buffer
        end;
      count--;
    end SELECT
  end loop
```

Clause is ***active*** only when condition is true

ACCEPT dequeues a client request and performs the operation

Single thread: no need for mutual exclusion

Non-deterministic choice between a set of ***guarded*** ACCEPT clauses

Message passing

- Dynamic invocations between threads can be thought of as general **message passing**
  - Thread X can send a message to Thread Y
  - Contents of message can be arbitrary data
- Can be used to build **Remote Procedure Call** (RPC)
  - Message includes name of operation to invoke along with as any parameters
  - Receiving thread checks operation name, and invokes the relevant code
  - Return value(s) sent back as another message
- (Called **Remote Method Invocation** (RMI) in Java)

We will discuss message passing and RPC in detail next term; a taster now, as these ideas apply to local, not just distributed, systems.

Message passing semantics

- Can conceptually view sending a message to be similar to sending an email:
  1. Sender prepares contents locally, and then sends
  2. System eventually delivers a **copy** to receiver
  3. Receiver checks for messages
- In this model, sending is **asynchronous**:
  - Sender doesn't need to wait for message delivery
  - (but he may, of course, choose to wait for a reply)
- Receiving is also asynchronous:
  - messages first **delivered** to a mailbox, later **retrieved**
  - message is a copy of the data (i.e. no actual sharing)

8

Message passing advantages

- **Copy semantics** avoid race conditions
  - At least directly on the data
- Flexible API: e.g.
  - **Batching**: can send *K* messages before waiting; and can similarly batch a set of replies
  - **Scheduling**: can choose when to receive, who to receive from, and which messages to prioritize
  - **Broadcast**: can send messages to many recipients
- Works both within and between machines
  - i.e. same design works for **distributed systems**
- Explicitly used as basis of some languages…

9

Example: Linda

- Concurrent programming language based on the abstraction of the **tuple space**
  - A [distributed] shared store which holds variable length typed tuples, e.g. "`('tag', 17, 2.34, 'foo')`"
  - Allows asynchronous "pub sub" messaging
- Processes can create new tuples, read tuples, or read-and-remove tuples

```
out(<tuple>);       // publishes tuple in TS
t = rd(<pattern>);  // reads a tuple matching pattern
t = in(<pattern>);  // as above, but removes tuple
```

- Weird… and difficult to implement efficiently

10

Example: occam

- Language based on Hoare's **Communicating Sequential Processes** (CSP) formalism
  - A "process algebra" for modeling concurrency
- Processes **synchronously** communicate via **channels**

```
<channel> ? <variable>     // an input process
<channel> ! <expression>   // an output process
```

- Build complex processes via SEQ, PAR and ALT, e.g.

```
ALT
  count1 < 100 & c1 ? Data
    SEQ
      count1:= count1 + 1
      merged ! data
  count2 < 100 & c2 ? Data
    SEQ
      count2:= count2 + 1
      merged ! data
```

11

Example: Erlang

- Functional programming language designed in mid 80's, made popular more recently
- Implements the **actor model**
- **Actors**: lightweight language-level processes
  - Can spawn() new processes very cheaply
- **Single-assignment**: each variable is assigned only once, and thereafter is immutable
  - But values can be sent to other processes
- **Guarded receives** (as in Ada, occam)
  - Messages delivered in order to local mailbox

Proponents of Erlang argue that lack of synchronous message passing prevents deadlock. Why might this claim be misleading?

Producer-Consumer in Erlang

```
-module(producerconsumer).
-export([start/0]).

start() ->
  spawn(fun() -> loop() end).

loop() ->
  receive
    {produce, item } ->
      enter_item(item),
      loop();
    {consume, Pid } ->
      Pid ! remove_item(),
      loop();
    stop ->
      ok
end.
```

Invoking start() will spawn an actor…

receive matches messages to patterns

explicit tail-recursion is required to keep the actor alive…

… so if send 'stop', process will terminate.

13

Message passing: summary

- A way of sidestepping (at least some of) the issues with shared memory concurrency
  - No direct access to data => no **data** race conditions
  - Threads choose actions based on message
- Explicit message passing can be awkward
  - Many weird and wonderful languages ;-)
- Can also use with traditional languages, e.g.
  - Transparent messaging via RPC/RMI
  - Scala, Kilim (actors on Java, or for Java), …

We have eliminated some of the issues associated with shared memory, but these are still concurrent programs subject to deadlock, livelock, etc.

Composite operations

- So far have seen various ways to ensure safe concurrent access to a single object
  - e.g. monitors, active objects, message passing
- More generally want to handle **composite operations**:
  - i.e. build systems which act on multiple distinct objects
- As an example, imagine an internal bank system which allows account access via three method calls:

```
int amount = getBalance(account);
bool credit(account, amount);
bool debit(account, amount);
```

- If each is thread-safe, is this sufficient?
  - Or are we going to get into trouble???

15

Composite operations

- Consider two concurrently executing client threads:
  - One wishes to transfer 100 quid from the savings account to the current account
  - The other wishes to learn the combined balance

```
// thread 1: transfer 100
// from savings->current
  debit(savings, 100);
  credit(current, 100);
```

```
// thread 2: check balance
  s = getBalance(savings);
  c = getBalance(current);
  tot = s + c;
```

- If we're unlucky then:
  - Thread 2 could see balance that's too small
  - Thread 1 could crash after doing debit() – ouch!
  - Server thread could crash at any point – ouch?

16

Problems with composite operations

Two separate kinds of problem here:

1. **Insufficient Isolation**
   - Individual operations being atomic is not enough
   - E.g., want the credit & debit making up the transfer to happen as one operation
   - Could fix this particular example with a new transfer() method, but not very general ...
2. **Fault Tolerance**
   - In the real-word, programs (or systems) can fail
   - Need to make sure we can recover safely

17

Transactions

- Want programmer to be able to specify that a set of operations should happen **atomically**, e.g.

```
// transfer amt from A -> B
transaction {
 if (getBalance(A) > amt) {
    debit(A, amt);
    credit(B, amt);
    return true;
  } else return false;
}
```

- A **transaction** either executes correctly (in which case we say it **commits**), or has no effect at all (i.e. it **aborts**)
  - <u>regardless of other transactions, or system crashes</u>!

18

ACID Properties

Want committed transactions to satisfy four properties:

- **Atomicity**: either all or none of the transaction's operations are performed
  - Programmer doesn't need to worry about clean up
- **Consistency**: a transaction transforms the system from one consistent state to another – i.e., preserves **invariants**
  - Programmer must ensure e.g. conservation of money
- **Isolation**: each transaction executes [as if] isolated from the concurrent effects of others
  - Can ignore concurrent transactions (or partial updates)
- **Durability**: the effects of committed transactions survive subsequent system failures
  - If system reports success, must ensure this is recorded on disk

This is a different use of the word "**atomic**" than previously; we will just have to live with that, unfortunately.

ACID Properties

Can group these into two categories

1. **Atomicity** & **Durability** deal with making sure the system is safe even across failures
   - (**A**) No partially complete txactions
   - (**D**) Transactions previously reported as committed don't disappear, even after a system crash
2. **Consistency** & **Isolation** ensure correct behavior even in the face of concurrency
   - (**C**) Can always code as if invariants in place
   - (**I**) Concurrently executing transactions are indivisible

20

Isolation

- To ensure a transaction executes in isolation could just have a server-wide lock... simple!

```
// transfer amt from A -> B
transaction {  // acquire server lock
 if (getBalance(A) > amt) {
    debit(A, amt);
    credit(B, amt);
    return true;
  } else return false;
}               // release server lock
```

- But doesn't allow any concurrency...
- And doesn't handle mid-transaction failure (e.g. what if we are unable to credit the amount to **B**?)

21

Isolation – Serialisability

- The idea of executing transactions **serially** (one after the other) is a useful **model for the programmer**:
  - To improve performance, **transaction systems** execute many transactions concurrently
  - But programmers must only observe behaviours consistent with a possible serial execution: **serialisability**
- Consider two transactions, **T1** and **T2**

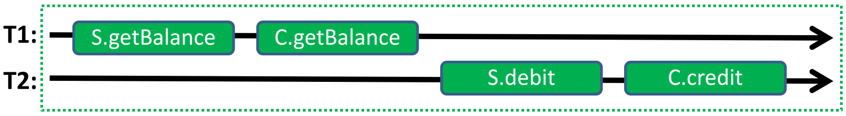
```
T1 transaction {
  s = getBalance(S);
  c = getBalance(C);
  return (s + c);
}
```

```
T2 transaction {
  debit(S, 100);
  credit(C, 100);
  return true;
}
```

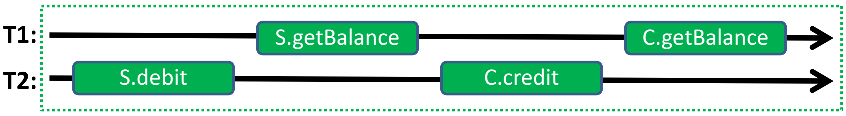
- If assume individual operations are atomic, then there are six possible ways the operations can interleave...

22

Isolation – serialisability



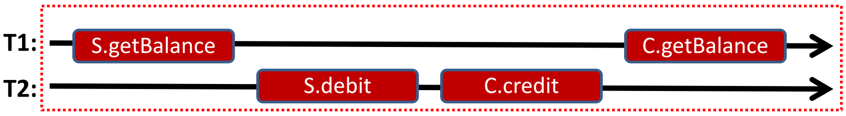
- First case is a **serial execution** and hence **serialisable**



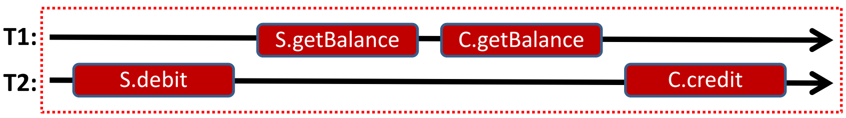
- Second case is **not serial** as transactions are interleaved
  - Its results are identical to serially executing **T2** and then **T1**
  - The schedule is therefore **serialisable**
- Informally: it is serialisable because we have only swapped the execution orders of **non-conflicting operations**
  - All of **T1**'s operations on any objects happen after **T2**'s update

23

Isolation – serialisability



- This execution is neither **serial** nor **serialisable**
  - **T1** sees inconsistent values: old **S** and new **C**



- This execution is also neither **serial** nor **serialisable**
  - **T1** sees inconsistent values: new **S**, old **C**
- Both orderings swap **conflicting operations** such that there is no matching serial execution

24

Conflict Serialisability

- There are many flavours of serialisability
- **Conflict serialisability** is satisfied for a schedule **S** if (and only if):
  - It contains the same set of operations as some serial schedule **T**; and
  - All **conflicting operations** are ordered the same way as in **T**
- Define **conflicting** as **non-commutative**
  - I.e., differences are permitted between the execution ordering and **T**, but they can't have a visible impact

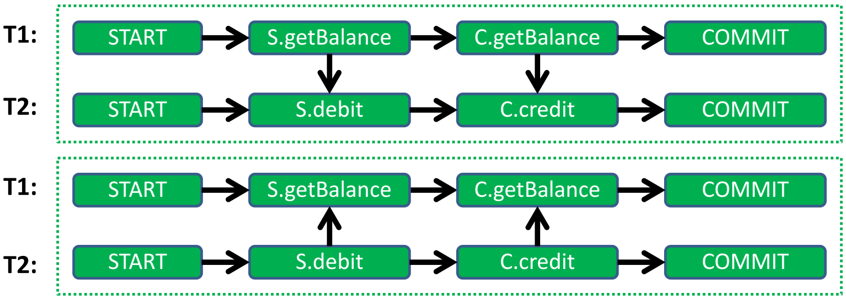
25

History graphs

- Can construct a graph for any execution schedule:
  - Nodes represent individual operations, and
  - Arrows represent "**happens-before**" relations
- Insert edges between operations within a given transaction in **program order** (i.e., as written)
- Insert edges between **conflicting** operations operating on the same objects, ordered by execution schedule
  - e.g. A.credit(), A.debit() commute [don't conflict]
  - A.credit() and A.addInterest() **do** conflict
- The next few graphs represent **specific execution schedules** rather than **possible schedules**

26

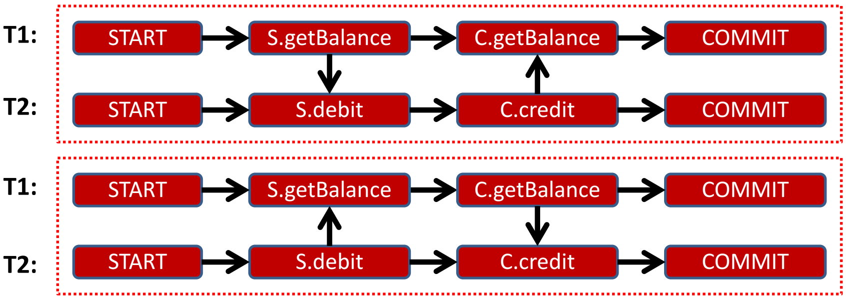
History graphs: good schedules



- Same schedules as before (both ok)
- Can easily see that everything in **T1** either happens before everything in **T2**, or vice versa
  - Hence schedule can be serialised

27

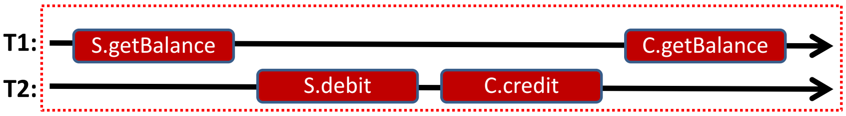
History graphs: bad schedules



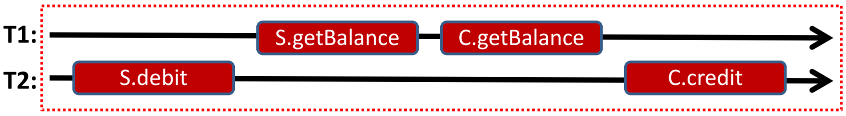
- Cycles indicate that schedules are bad :-(
- Neither transaction strictly "happened before" the other:
  - Arrows from **T1** to **T2** mean "**T1** must happen before **T2**"
  - But arrows from **T2** to **T1** => "**T2** must happen before **T1**"
- Can't both be true → schedules are **non-serialisable**

28

Isolation – serialisability



- This execution is neither **serial** nor **serialisable**
  - **T1** sees inconsistent values: old **S** and new **C**



The **transaction system** must ensure that, regardless of actual concurrent execution used to improve performance, only **serialisable orderings** are visible to the **transaction programmer**.

Summary + next time

- Concurrency without shared data (Active Objects)
- Message passing, actor model (Linda, occam, Erlang)
- Composite operations; transactions; ACID properties
- Isolation and serialisability
- History graphs; good (and bad) schedules

- Next time – more on transactions:
  - Isolation vs. strict isolation; enforcing isolation
  - Two-phase locking; rollback
  - Timestamp ordering (TSO); optimistic concurrency control (OCC)
  - Isolation and concurrency summary