Concurrent Systems
8L for Part IB

Handout 3

Dr Robert Watson

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…

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 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

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)

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)

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…

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

Example: occam

- Language based on Hoare's 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
```

Example: Erlang

- Functional programming language designed in mid 80's, made popular more recently
- **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

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.

Message Passing: Summary

- A way of sidestepping (at least some of) the issues with shared memory concurrency
  - No direct access to data => no 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), …

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???

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?

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

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**)
    - regardless of other transactions, or system crashes!

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
  - 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

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**) Txactions 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 txactions are invisible

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?)

Isolation – Serializability

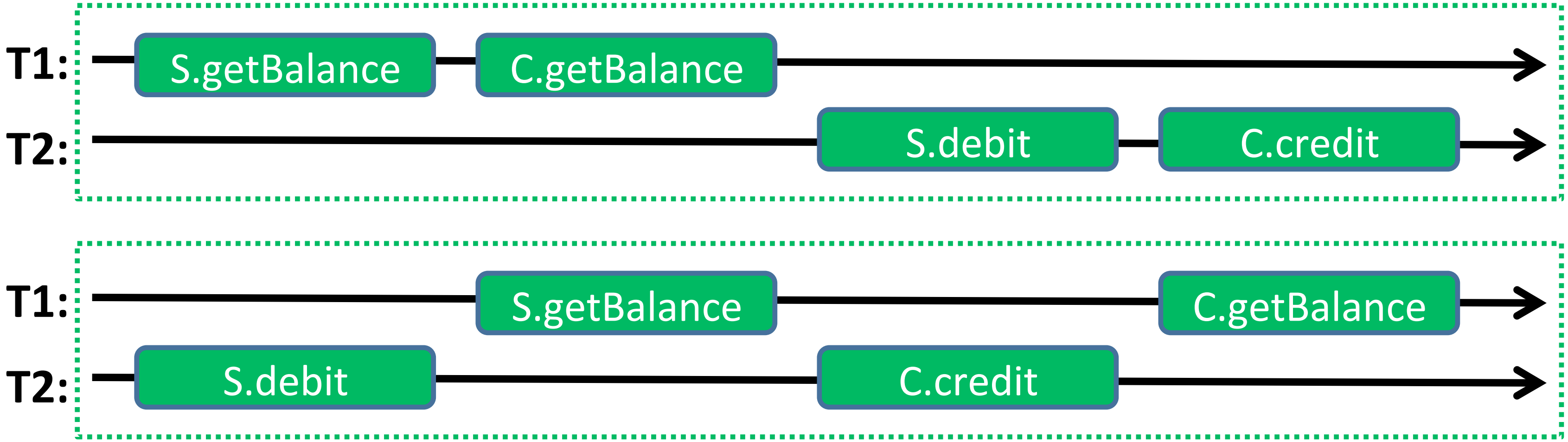
- The idea of executing transactions **serially** (one after the other) is a useful model
  - We want to run transactions concurrently
  - But the result should be **as if** they ran serially
- Consider two transactions, T1 and T2

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

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

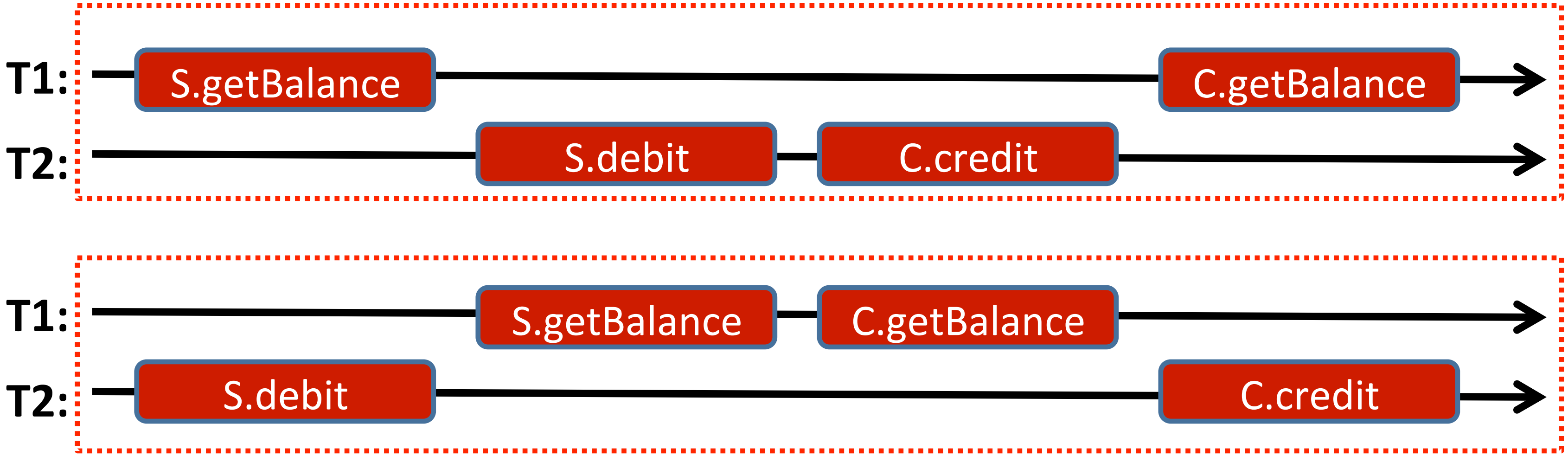
- If assume individual operations are atomic, then there are six possible ways the operations can interleave…

Isolation – Serializability



- First case is serial and, as expected, all ok
- Second case is not serial … but result is fine
  - – Both of T1's operations happen after T2's update
  - – This is a **serializable** schedule [as is first case]

Isolation – Serializability



T1: S.getBalance C.getBalance

T2: S.debit C.credit

T1: S.getBalance C.getBalance

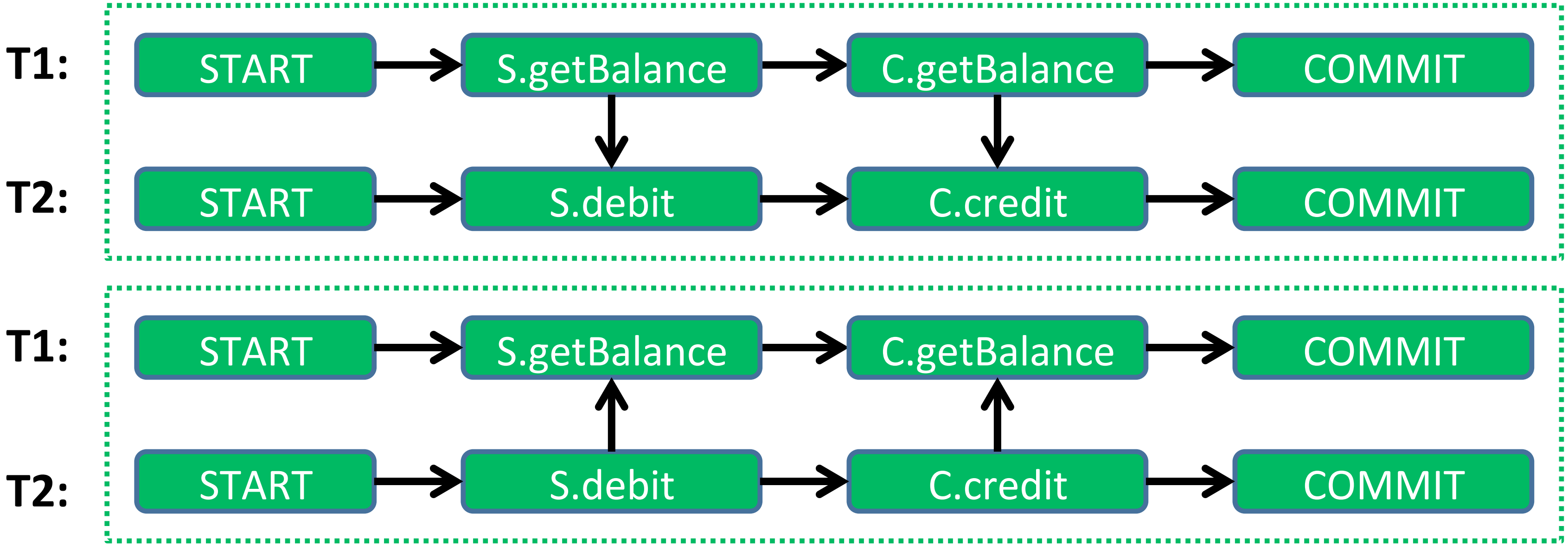
T2: S.debit C.credit

- Neither of these two executions is ok
- T1 sees inconsistent values:
  - (top) sees updated version of C, but old version of S
  - (bottom) sees updated S, but original version of C

History Graphs

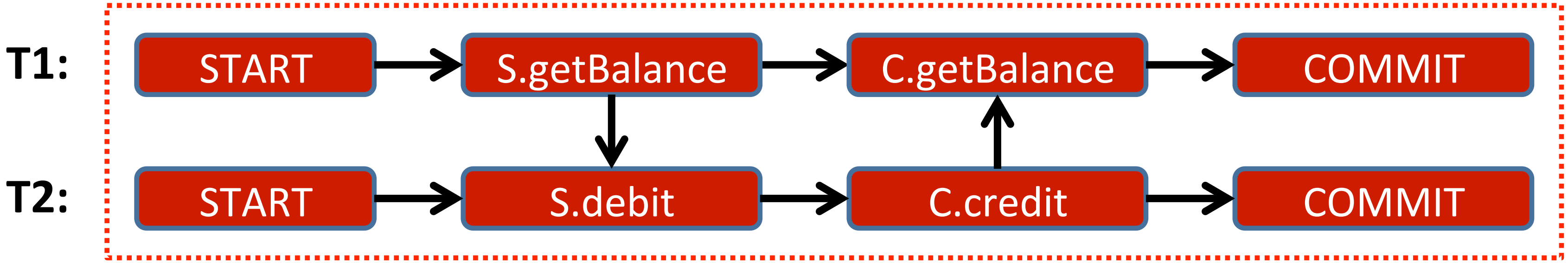
- Can construct a graph for any execution:
  - Nodes represent individual operations, and
  - Arrows represent "happens-before" relations
- Operations within a given transaction must happen in program order (i.e. as written)
- **Conflicting** operations are ordered by the implementation of the underlying object
  - conflicting operations = non-commutative
  - e.g. A.credit(), A.debit() commute [don't conflict], while A.credit() and A.addInterest() **do** conflict

History Graphs: Good Schedules



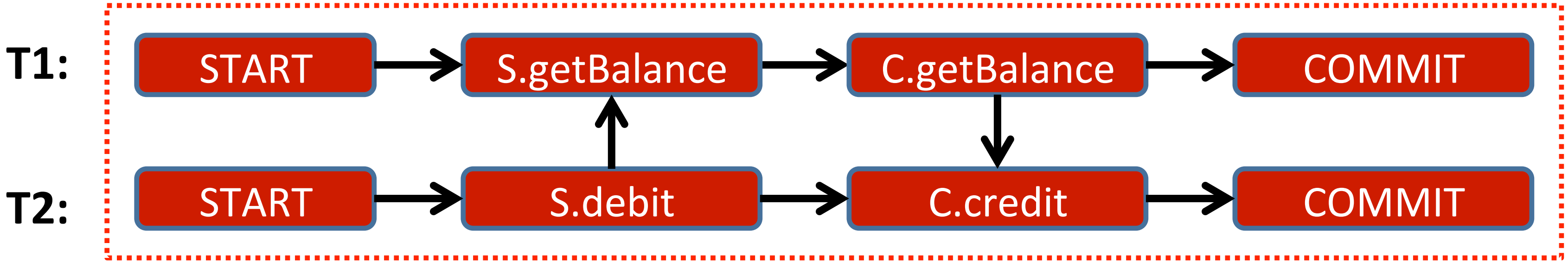
- Same schedules as before (both ok)
- Can easily see that everything in T1 either happens before everything in T2, or vice versa
  - Hence schedule can be serialized

History Graphs: Bad Schedules



T1: START → S.getBalance → C.getBalance → COMMIT

T2: START → S.debit → C.credit → COMMIT



T1: START → S.getBalance → C.getBalance → COMMIT

T2: START → S.debit → C.credit → COMMIT

- Both schedules are bad :-(
  - 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 not serializable.

Causes of Bad Schedules

- **Lost Updates**
  - T1 updates (writes) an object, but this is then overwritten by concurrently executing T2
  - (also called a write-write conflict)
- **Dirty Reads**
  - T1 reads an object which has been updated an uncommitted transaction T2
  - (also called a read-after-write conflict)
- **Unrepeatable Reads**
  - T1 reads an object which is then updated by T2
  - Not possible for T1 to read the same value again
  - (also called a write-after-read conflict)

Isolation and Strict Isolation

- Ideally want to avoid all three problems
- Two ways: Strict Isolation and Non-Strict Isolation
  - **Strict Isolation**: guarantee we never experience lost updates, dirty reads, or unrepeatable reads
  - **Non-Strict Isolation**: let transaction continue to execute despite potential problems
- Non-strict isolation usually allows more concurrency but can lead to complications
  - e.g. if T1 reads something written by T2 (a "dirty read") then T1 cannot commit until T2 commits
  - and T1 must abort if T2 aborts: **cascading aborts**

Enforcing Isolation

- In practice there are a number of techniques we can use to enforce isolation (of either kind)
- We will look at:
  - Two-Phase Locking (2PL);
  - Timestamp Ordering (TSO); and
  - Optimistic Concurrency Control (OCC)

Two Phase Locking (2PL)

- Associate a lock with every object
  - Could be mutual exclusion, or MRSW
- Transactions proceed in two phases:
  - Expanding Phase: during which locks are acquired but none are released
  - Shrinking Phase: during which locks are released, and no more are acquired
- Operations on objects occur in either phase, providing appropriate locks are held
  - Should ensure serializable execution

2PL Example

Expanding Phase

Shrinking Phase

```
// transfer amt from A -> B
transaction {
 readLock(A);
 if (getBalance(A) > amt) {
    writeLock(A);
    debit(A, amt);
    writeLock(B);
    credit(B, amt);
    writeUnlock(B);
    addInterest(A);
    writeUnlock(A);
    tryCommit(return=true);
  } else {
    readUnlock(A);
    tryCommit(return=false);
}
```

Acquire a read lock (shared) before ‘read’ A

Upgrade to a write lock (exclusive) before write A

Acquire a write lock (exclusive) before write B

Release locks when done to allow concurrency

Problems with 2PL

- Requires knowledge of which locks required
  - Can be automated in many systems
- Risk of deadlock
  - Can attempt to impose a partial order
  - Or can detect deadlock and abort, releasing locks
  - (this is safe for transactions, which is nice)
- Non-strict Isolation: releasing locks during execution means others can access those objects
  - e.g. T1 updates A, then releases write lock; now T2 can read or overwrite the uncommitted value
  - Hence T2's fate is tied to T1 (whether commit or abort)
  - Can fix with **strict 2PL**: hold all locks until transaction end

Strict 2PL Example

```
// transfer amt from A -> B
transaction {
 readLock(A);
 if (getBalance(A) > amt) {
    writeLock(A);
    debit(A, amt);
    writeLock(B);
    credit(B, amt);
    addInterest(A);
    tryCommit(return=true);
  } else {
    readUnlock(A);
    tryCommit(return=false);
} on commit, abort {
    unlock(A);
    unlock(B);
}
```

Expanding Phase

Unlock All Phase

Retain lock on B here to ensure **strict isolation**