Learning Objectives

- Understand locks and be able to recognize when locks are required
- Be aware of deadlocks and deadlock mitigation
- Understand the importance of transaction processing systems and a TP system’s multi-tiered architecture
- Be able to describe two phase locking (2PL)
- Be able to describe the two phase commit protocol (2PC)
- Understand how system availability trades off with system consistency as described by the CAP theorem
Transaction Processing (TP) Systems

• Historically, one of the first was American Airlines SABRE – Semi-Automated Business Research Environment - 83,000 transactions per day (1960’s)
• Became IBM’s Airline Control Program
• Became IBM’s Transaction Processing Facility (TPF)
• Many such modern systems exist:
  • Oracle Tuxedo (Thousands of transactions per second)
  • IBM’s Customer Information Control System (CICS)
  • Most databases and messaging systems provide for transactional support
  • JEE and Microsoft .NET both provide extensive capabilities for creating and deploying TP applications
TP System Architecture

This represents a multi-tiered TP application. The user-device might be a gas pump or retail sales terminal or browser.

Adapted From "Principles Of Transaction Processing" Bernstein and Newcomer
Transactions (ACID)

- **Atomic**: All or nothing. No intermediate states are visible. No possibility that only part of the transaction ran. If a transaction fails or aborts prior to committing, the TP system will undo the effects of any updates (will recover). We either commit or abort the entire process. Checkpointing and Logging and recoverable objects can be used to ensure a transaction is atomic with respect to failures.

- **Consistent**: system invariants preserved, e.g., if there were n dollars in a bank before a transfer transaction then there will be n dollars in the bank after the transfer. This is largely in the hands of the application programmer.

- **Isolated**: Two transactions do not interfere with each other. They appear as serial executions. This is the case even though transactions may run concurrently. Locking is often used to prevent one transaction from interfering with another.

- **Durable**: The commit causes a permanent change to stable storage. This property may be obtained with log-based recovery algorithms. If there has been a commit but updates have not yet been completed due to a crash, the logs will hold the necessary information on recovery.
Assume concurrent visits

private double balance;  // This is all that is required for many applications. But TP middleware must do much more.

public synchronized void deposit(double amount) throws RemoteException {
    balance = balance + amount;
}

public synchronized void withdraw(double amount) throws RemoteException {
    balance = balance - amount;
}

If one thread invokes a method it acquires a lock. Another thread will be blocked until the lock is released.

What happens if we don’t synchronize?
Consider a shared queue and two operations:

```java
synchronized first() { if Q is empty return false
    else remove and return front
}

synchronized append() { add to rear }
```

Is this sufficient?
No. If the queue is empty the client of first() will have to poll on the method. What’s wrong with polling?

It is also potentially unfair. Why?
Consider again the shared queue and two operations:

```java
synchronized first() {
    if queue is empty call wait()
    remove from front
}
synchronized append() {
    adds to rear
    call notify()
}
```

When threads can synchronize their actions on an object by means of `wait` and `notify`, the server holds on to requests that cannot immediately be satisfied and the client waits for a reply until another client has produced whatever they need.

Note that both methods are synchronized. Only one thread at a time is allowed in.

This is a simple example. Wait/notify gets tricky fast.
Back to Transactions

- A client may require that a sequence of separate requests to a single server be isolated and atomic.
  - Isolated => Free from interference from other concurrent clients.
  - Atomic => Either all of the operations complete successfully or they have no effect at all in the presence of server crashes.
  - We also want serializability => If two transactions T1 and T2 are running, we want it to appear as if T1 was followed by T2 or T2 was followed by T1.
  - But, interleaving may have occurred (we like interleaving for performance reasons).
Assume each operation on the server is synchronized - Happy Case

Client 1 Transaction T;
  a.withdraw(100);
  b.deposit(100);
  c.withdraw(200);
  b.deposit(200);

Client 2 Transaction W;
  total = x.getBalance();
  total = total +
  y.getBalance();
  total = total +
  z.getBalance();

Suppose both run to completion (no partial execution) => atomic.

Why are we isolated?
Assume each operation on the server is synchronized

Client 1 Transaction T;
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

Suppose both run to completion (no partial execution) => atomic.

Client 2 Transaction W;
total = a.getBalance();
total = total + b.getBalance();
total = total + c.getBalance();

Are we isolated?
Assume each operation on the server is synchronized

Client 1 Transaction T;
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

Client 2 Transaction W;
total = a.getBalance();
total = total +
    b.getBalance();
total = total +
    c.getBalance();

Inconsistent retrieval!
Assume each operation on the server is synchronized

Client 1 Transaction T;
bal = b.getBalance();
b.setBalance(bal*1.1);

Suppose both run to completion with no partial execution => Atomic.

Client 2 Transaction W;
bal = b.getBalance();
b.setBalance(bal*1.1);

But are we isolated?
Assume each operation on the server is synchronized

Client 1 Transaction T;
bal = b.getBalance()
b.setBalance(bal*1.1);

Client 2 Transaction W;
bal = b.getBalance();
b.setBalance(bal*1.1);

Lost Update!
Assume each operation on the server is synchronized

Transaction T;

a. withdraw(100);
b. deposit(100);
c. withdraw(200);
b. deposit(200);

The aim of any server that supports transactions is to maximize concurrency. So, transactions are allowed to execute concurrently if they would have the same effect as serial execution.

Locking is the most popular mechanism to achieve transaction Isolation.

Each transaction is created and managed by a coordinator.
Interacting with a coordinator

Transaction T

tid = openTransaction();
    a.withdraw(tid,100);
    b.deposit(tid,100);
    c.withdraw(tid,200);
    b.deposit(tid,200);
closeTransaction(tid) or
abortTransaction(tid)

Coordinator Interface:

call openTransaction() to get transID

call closeTransaction(transID) to commit or abort

call abortTransaction(TransID)
Serially Equivalent

• For two transactions to be *serially equivalent*, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access. (Coulouris)

• Let $r_1(x)$ mean that transaction 1 reads $x$.
• Let $w_2(x)$ mean that transaction 2 writes $x$.
• $r_1(x)$ $r_2(x)$ do not conflict (may be ordered either way).
• $r_1(x)$ $w_2(x)$ do conflict (order matters)
• $w_1(x)$ $w_2(x)$ do conflict (order matters)
Locking to Attain Serializability

• With locks, each transaction reserves access to the data it uses.
• There are read locks rL₁(x) and write locks wL₂(x).
• Before reading, a read lock is set. Before writing, a write lock is set.
• A transaction can obtain a lock only if no other transaction has a conflicting lock on the same data item.
• Locks may be removed with wU₁(y) or rU₁(x)
• In the next two slides, we return to the earlier examples and apply this locking scheme. They become serially equivalent.
Allow either one to run first - Two Phase Locking for Concurrency Control

Client 1 Transaction $T$;
Get write lock on $a$
$a$.withdraw(100);
Get write lock on $b$
b.deposit(100);
Get write lock on $c$
c.withdraw(200);
b.deposit(200);
Unlock $a,b,c$

Client 2 Transaction $W$;
Get a read lock on $a$
total = $a$.getBalance();
Get a read lock on $b$
total = total +
  b.getBalance();
Get a read lock on $c$
total = total +
  c.getBalance();
Unlock $a,b,c$
Allow either one to run first - Two Phase Locking for Concurrency Control

Client 1 Transaction T:
Get a read lock on b
bal = b.getBalance()
Upgrade to write
b.setBalance(bal*1.1);
Unlock b

Client 2 Transaction W:
Get a read lock on b
bal = b.getBalance();
Upgrade to write lock
b.setBalance(bal*1.1);
Unlock b
Locking is not enough

- Transaction T1
  - rL_1(x)
  - r_1(x)
  - rU_1(x)
  - wL_1(y)
  - w_1(y)
  - wU_1(y)

- Transaction T2
  - rL_2(y)
  - r_2(y)
  - wL_2(x)
  - w_2(x)
  - rU_2(y)
  - wU_2(x)
What pairs are in conflict?

- Transaction T1
  - rL₁(x)
  - r₁(x)
  - rU₁(x)
  - wL₁(y)
  - w₁(y)
  - wU₁(y)

- Transaction T2
  - rL₂(y)
  - r₂(y)
  - wL₂(x)
  - w₂(x)
  - rU₂(y)
  - wU₂(x)

“all pairs of conflicting operations of the two transactions be executed in the same order”

Coulouris
What pairs are in conflict?

- Transaction T1
  - rL₁(x)
  - r₁(x)
  - rU₁(x)
  - wL₁(y)
  - w₁(y)
  - wU₁(y)

- Transaction T2
  - rL₂(y)
  - r₂(y)
  - wL₂(x)
  - w₂(x)
  - rU₂(y)
  - wU₂(x)

To be serially equivalent:
- If r₁(x) occurs before w₂(x), then w₁(y) must occur before r₂(y) and
- If r₁(x) occurs after w₂(x), then w₁(y) must occur after r₂(y).
Locking is not enough

- Transaction T1
  - $rL_1(x)$
  - $r_1(x)$
  - $rU_1(x)$
  - $wL_1(y)$
  - $w_1(y)$
  - $wU_1(y)$

- Transaction T2
  - $rL_2(y)$
  - $r_2(y)$
  - $wL_2(x)$
  - $w_2(x)$
  - $rU_2(y)$
  - $wU_2(x)$

Locking alone does not enforce the rules on serially equivalence.
Locking is not enough

- Transaction T1
  - rL₁(x)
  - r₁(x)
  - rU₁(x)
  - wL₁(y)
  - w₁(y)
  - wU₁(y)

- Transaction T2
  - rL₂(y)
  - r₂(y)
  - wL₂(x)
  - w₂(x)
  - rU₂(y)
  - wU₂(x)

We now have r₁(x), r₂(y), w₂(x), w₁(y). But we need...
Locking is not enough

We now have \( r_1(x), r_2(y), w_2(x), w_1(y) \). But we need…

either T1 followed by T2 or T2 followed by T1.

We need either \( r_1(x), w_1(y), r_2(y), w_2(x) \) or \( r_2(y), w_2(x), r_1(x), w_1(y) \).

How do we guarantee that?

**Two phase locking (2PL)** demands that all locks are obtained for each transaction before releasing any of them!
Lock and Unlock in two Phases

- Transaction T1                 Transaction T2

  rL₁(x)                            rL₂(y)                      Now T1 and T2 are serialized.
  r₁(x)                            r₂(y)              This may lead to deadlock. The serialization proof exists but is beyond course scope.
  wL₁(y)                            wL₂(x)
  w₁(y)                            w₂(x)
  wU₁(y)                            rU₂(y)
  rU₁(x)                            wU₂(x)
What might Lock_Item() look like?

```
Lock_Item(x)
    B: if(Lock(x) == 0)
        Lock(x) = 1
    else {
        wait until Lock(x) == 0 and we are woken up.
        GOTO B
    }
Now, a transaction is free to use x.
```

Not interleaved with other code until this terminates or waits. In java, this would be a synchronized method.

Similar to the code above that used a shared queue.
And unlock_item()?

The transaction is done using x.

Unlock_Item(x)
Lock(x) = 0
if any transactions are waiting then
wake up one of the waiting transactions.

Not interleaved with other code. If this were java, this method would be synchronized.
Does this allow for any concurrency?

Transaction $T_1$  Transaction $T_2$

Lock_Item(x)     Lock_Item(y)
$T_1$ uses x           $T_2$ uses y
Unlock_Item(x) Unlock_Item(y)

If x differs from y these two transactions proceed concurrently.
If both want to use x, one waits until the other completes.

In reality, the coordinator would do the locking.
Locks May Lead to Deadlock

Four Requirements for deadlock:

(1) Resources need mutual exclusion. They are not thread safe.
(2) Resources may be reserved while a process is waiting for more.
(3) Preemption is not allowed. You can't force a process to give up a resource.
(4) Circular wait is possible. X wants what Y has and Y wants what Z has but Z wants what X has.

Solutions (short course):

Prevention (disallow one of the four)
Avoidance (study what is required by all before beginning)
Detection (using time-outs or wait-for graphs) and recovery
# Deadlock

<table>
<thead>
<tr>
<th>Transaction $T$</th>
<th>Transaction $U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>Operations</td>
</tr>
<tr>
<td>$a.deposit(100)$</td>
<td>$b.deposit(200)$</td>
</tr>
<tr>
<td>$b.withdraw(100)$</td>
<td></td>
</tr>
<tr>
<td>***</td>
<td>waits for $U$'s</td>
</tr>
<tr>
<td></td>
<td>lock on $B$</td>
</tr>
<tr>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>***</td>
<td>***</td>
</tr>
</tbody>
</table>

Local Transactions (Single Server)

- Typically handled directly by a database.
- Call `beginTransaction` on an `SQLConnection` object.
- Execute SQL statements.
- Call `commit` or `rollback`.
- Locks may be held on the rows/tables involved.
- Everything is done on a copy until the commit or rollback.
- Deadlock may be detected with wait-for graphs.
- In distributed transactions, deadlock may be detected with time-outs.
Transactions On Objects (Single Server)

a = lookUp("A")
b = lookUp("B")
beginTran
  x = a.read()
b.write(x)
closeTran or abortTran

a = lookUp("A")
beginTran
  x = a.read()
closeTran or abortTran

Lock management, recovery management and traditional middleware

Recoverable objects
What is a Recoverable Object?

A recoverable object follows the Golden Rule of Recoverability:

“Never modify the only copy.”

The transaction make changes to local copies of resources until a commit or rollback.

If a transaction fails, only the tentative versions of the objects have changed, not the non-volatile copy.

When the server is running it can keep all of its objects in its volatile memory and records its committed objects in a recovery file.

Upon recovery, the server can restore the object’s latest committed versions.
Distributed Transactions (More than one server)

Begin transaction BookTrip
book a plane from Qantas
book hotel from Hilton
book rental car from Hertz
End transaction BookTrip

The Two Phase Commit Protocol is a classic solution for atomicity and consistency.
Interacting with a coordinator

Transaction T

tid = openTransaction();

a.withdraw(tid,100);
b.deposit(tid,100);
c.withdraw(tid,200);
b.deposit(tid,200);

closeTransaction(tid) or
abortTransaction(tid)

Coordinator Interface:

openTransaction() -> transID
closeTransaction(transID) ->
commit or abort
abortTransaction(TransID)

Think about atomicity and
consistency.
Client Talks to a Coordinator

Any server

BookTrip Coordinator

BookTrip Client

openTrans

Unique Transaction ID

TID

Different servers

BookPlane Participant
Recoverable objects needed to book a plane

BookHotel Participant
Recoverable objects needed to book a hotel.

BookRentalCar Participant
Recoverable objects needed to rent a car.

TID = openTransaction()
Client Uses Services

- **Any server**
  - BookTrip
  - Coordinator

- **Different servers**
  - BookPlane Participant
    - Recoverable objects needed to book a plane
  - BookHotel Participant
    - Recoverable objects needed to book a hotel.
  - BookRentalCar Participant
    - Recoverable objects needed to rent a car.

- **BookTrip Client**
  - Call + TID
  - plane.bookFlight(111, ”Seat32A”, TID)
**Participants Talk to Coordinator**

- **BookTrip Coordinator**
  - `join(TID, ref to participant)`

- **BookTrip Client**
  - The participant knows where the coordinator is because that information can be included in the TID (e.g., an IP address.)
  - The coordinator now has a pointer to the participant.

- **Different servers**
  - The participant only calls `join` if it has not already done so.

- **BookPlane Participant**
  - Recoverable objects needed to book a plane

- **BookHotel Participant**
  - Recoverable objects needed to book a hotel.

- **BookRentalCar Participant**
Suppose All Goes Well (1)

Different servers

BookPlane Participant
Recoverable objects needed to book a plane

BookHotel Participant
Recoverable objects needed to book a hotel.

BookRentalCar Participant
Recoverable objects needed to rent a car.
Suppose All Goes Well (2)

- **BookTrip Coordinator**
  - Coordinator begins 2PC and this results in a GLOBAL COMMIT sent to each participant.

- **BookTrip Client**
  - OK returned
  - OK returned
  - OK returned
  - OK returned

- **Different servers**
  - **BookPlane Participant**
    - Recoverable objects needed to book a plane
  - **BookHotel Participant**
    - Recoverable objects needed to book a hotel.
  - **BookRentalCar Participant**
    - Recoverable objects needed to rent a car.

- **CloseTransaction(TID) Called**
This Time No Cars Available (1)

Different servers
BookPlane Participant
Recoverable objects needed to book a plane
BookHotel Participant
Recoverable objects needed to book a hotel.
BookRentalCar Participant
Recoverable objects needed to rent a car.

BookTrip Coordinator

BookTrip Client
OK returned
OK returned
NO CARS AVAIL
abortTransaction(TID) called
This Time No Cars Available (2)

**BookTrip Coordinator**
- Coordinator sends a `GLOBAL_ABORT` to all participants

**BookTrip Client**
- OK returned
- OK returned
- NO CARS AVAIL
- `abortTransaction(TID)` called

**Different servers**

**BookPlane Participant**
- Recoverable objects needed to book a plane

**BookHotel Participant**
- Recoverable objects needed to book a hotel.

**BookRentalCar Participant**
- Recoverable objects needed to rent a car.
This Time No Cars Available (3)

BookTrip Coordinator

AbortTransaction

Each participant Gets a GLOBAL_ABORT

Different servers

BookPlane Participant

ROLLBACK CHANGES

BookHotel Participant

ROLLBACK CHANGES

BookRentalCar Participant

ROLLBACK CHANGES

BookTrip Client

OK returned

OK returned

NO CARS AVAIL

AbortTransaction(TID)
BookPlane Server Crashes After Returning ‘OK’ (1)

Different servers

BookPlane Participant
Recoverable objects needed to book a plane

BookHotel Participant
Recoverable objects needed to book a hotel.

BookRentalCar Participant
Recoverable objects needed to rent a car.

BookTrip Coordinator

BookTrip Client

OK returned

OK returned

OK returned

OK returned

95-702 Transactions 45
BookTrip Server Crashes After Returning ‘OK’ (2)

BookTrip Coordinator

Coordinator executes 2PC:
- Ask everyone to vote.
- No news from the BookPlane Participant so multicast a GLOBAL ABORT

BookTrip Client

OK returned
OK returned
OK returned
OK returned

CloseTransaction(TID) Called

Different servers

BookPlane Participant
Recoverable objects needed to book a plane

BookHotel Participant
Recoverable objects needed to book a hotel.

BookRentalCar Participant
Recoverable objects needed to rent a car.
BookPlane Server Crashes after returning ‘OK’ (3)

Different servers

BookPlane Participant
Recoverable objects needed to book a plane

BookHotel Participant
ROLLBACK

BookRentalCar Participant
ROLLBACK

GLOBAL ABORT

BookTrip Coordinator

CloseTransaction(TID) Called

BookTrip Client

OK returned
OK returned
OK returned

ROLLBACK
Two-Phase Commit Protocol

Phase 1 BookTrip coordinator sends a Vote_Request to each process. Each process returns a Vote_Commit or Vote_Abort.
**Phase 2** BookTrip coordinator checks the votes. If every process votes to commit then so will the coordinator. In that case, it will send a Global_Commit to each process. If any process votes to abort the coordinator sends a GLOBAL_ABORT. Each process waits for a Global_Commit message before committing its part of the transaction.
2PC Finite State Machine from Tanenbaum

**BookTrip Coordinator**
- **Init**
  - Commit
  - Vote-request
  - Vote-abort
  - Global-abort
  - Global-commit
- **wait**
  - Vote-commit
  - Global-commit
- **Abort**
- **Commit**

**Participant**
- **Init**
  - Vote-request
  - Vote-abort
  - Vote-commit
- **Ready**
  - Global-commit
  - Global-abort
  - ACK
- **Abort**
  - 95-702 Transactions
- **Commit**
  - 50

State has already been saved to permanent storage.
2PC Blocks in Three Places

If waiting too long for a Vote-Request send a Vote-Abort
2PC Blocks in Three Places

If waiting too long
After Vote-request
Send a Global-Abort

Commit

Vote-request

Vote-abort

Global-abort

Abort

Commit

Vote-commit

Global-commit

Vote-request

Vote-commit

Vote-abort

Global-abort

Global-commit

ACK

ACK

Commit

Init

Ready

95-702 Transactions

52
2PC Blocks in Three Places

If waiting too long we can’t simply abort! We must wait until the coordinator recovers. We might also make queries on other participants.

---

If waiting too long we can’t simply abort! We must wait until the coordinator recovers. We might also make queries on other participants.
If this process learns that another has committed then this process is free to commit. The coordinator must have sent out a Global-commit that did not get to this process.
2PC Blocks in Three Places

If this process learns that another has aborted then it too is free to abort.
Suppose this process learns that another process is still in its init state. The coordinator must have crashed while multicasting the Vote-request. It’s safe for this process (and the queried process) to abort.
Tricky case: If the queried processes are all still in their ready state what do we know? We have to block and wait until the Coordinator recovers.
Summary 2PL and 2PC

• Two phase locking (2PL) is a concurrency control protocol – guarantees transactions have the same effect as serial execution. Guarantees transactions do not interfere with each other. Does it prevent deadlock?
• The two phase commit protocol (2PC) is an atomic commitment protocol – a special type of consensus protocol. Does it prevent deadlock? Typically used for distributed transactions.
• A consensus protocol tries to reach agreement in the presence of crashes or failures. All agree?
DS Principles For Internet Scale applications
(Ian Gorton SEI)

• **Goal**: highly available and highly scalable.
• **Principle**: Complex systems do not scale. 2PC is complex. Thus, on the internet, weak consistency is replacing strong consistency. For example, a change to a Google Doc may not be available to all immediately.
• **Principle**: Statelessness. Any server, any time implies keeping state off the server. No routing to correct server.
• **Principle**: Allow failures but be able to monitor system behavior.
• The **CAP Theorem** was first announced in 2000.
• **Principle**: Mixed approaches (consistency tradeoffs) are possible.
The CAP Theorem (Brewer)

• **Consistency**: if a value is written to node 1 then it is that value that is read from node 2. It is not the case that one node has an old value and another node has the most recent value. Brewer says “consistency (C) is equivalent to having a single up-to-date copy of the data;”

• **Available**: The system is highly available for updates.

• **Partition tolerance**: If a network failure occurs the system will still work.

• **CAP Theorem**: You may only have a system with two of these three. You may have either CA, CP, or AP.
The Traditional CAP Theorem (Brewer)

• Essentially, the CAP theorem says that in the face of network partitions, you may either have availability or consistency but not both.

• Example: Suppose there is a break in the network between an automated teller machine and the main banking database. We have a choice. We can be either unavailable for ATM use or we can allow for small transactions and be a little inconsistent. If we choose the latter we can still reconcile any inconsistencies later.

• Example: Suppose we are using HTML5 local storage and do some disconnected work offline. We are choosing availability over consistency – and may only require eventual consistency.
CAP Theorem Update

Brewer says (2012):

“Because partitions are rare, CAP should allow perfect C and A most of the time, but when partitions are present or perceived, a strategy that detects partitions and explicitly accounts for them is in order. This strategy should have three steps: detect partitions, enter an explicit partition mode that can limit some operations, and initiate a recovery process to restore consistency and compensate for mistakes made during a partition.”