Synchronization via Transactions

Concurrency Quiz

If two threads execute this program concurrently, how many different final values of X are there?

Initially, X == 0.

Thread 1

```
void increment() {
   int temp = X;
   temp = temp + 1;
   X = temp;
}
```

Thread 2

```
void increment() {
   int temp = X;
   temp = temp + 1;
   X = temp;
}
```

Answer:
A. 0
B. 1
C. 2
D. More than 2

Schedules/Interleavings

- Model of concurrent execution
- Interleave statements from each thread into a single thread
- If **any** interleaving yields incorrect results, some synchronization is needed

Thread 1

```
tmp1 = X;
tmp1 = tmp1 +
X = tmp1;
```

Thread 2

```
X;
tmp2 + 1;
p2;
```

```
tmp1 = X;
tmp2 = X;
tmp2 = tmp2 + 1;
tmp1 = tmp1 + 1;
X = tmp1;
X = tmp2;
```

If X==0 initially, X == 1 at the end. WRONG result!

Locks fix this with Mutual Exclusion

```
void increment() {
   lock.acquire();
   int temp = X;
   temp = temp + 1;
   X = temp;
   lock.release();
}
```

- Is mutual exclusion really what we want? *Don't we just want the correct result?*
- Some interleavings may give the correct result. *Why can't we keep these?*

Providing atomicity and isolation directly

- Critical regions need atomicity and isolation
- Definition: An atomic operation's effects either all happen or none happen.
  - Money transfer either debits one acct and credits the other, or no money is transferred
- Definition: An isolated operation is not affected by concurrent operations.
  - Partial results are not visible
  - This allows isolated operations to be put in a single, global order

Providing atomicity and isolation directly

- Implementing atomicity and isolation
  - Changes to memory are buffered (isolation)
  - Other processors see old values (isolation)
  - If something goes wrong (e.g., exception), system rolls back state to start of critical section (atomicity)
  - When critical region ends, changes become visible all at once (atomicity)
- Hardware
  - Processor support for buffering and committing values
- Software
  - Runtime system buffers and commits values

Transactions

- Transaction begin (xbegin)
  - Start of critical region
- Transaction end (xend)
  - End of critical region
- xbegin/xend can be implicit with atomic{}
- Transaction restart (or abort)
  - User decides to abort transaction
  - In Java throwing an exception aborts the transaction

```
atomic {
   acctA -= 100;
   acctB += 100;
}
```

Transaction to transfer $100 from acctA to acctB.

Atomicity and Isolation

- AcctA starts with $150
- Different blocks to update balance
  - Overnight batch process to read/process/write accounts
    - Debit $100
  - Telephone transaction to read/process/write quickly
    - Debit $90
- Isolation guarantees that phone update is not lost
  - It is allowed by atomicity
  - In fact, both transactions (in either order) should result in overdraft
    - AcctA = -$40

Atomicity and Isolation

- AcctA starts with $150
- Different blocks to update balance
  - Overnight batch process to read/process/write accounts
    - Debit $100
  - Telephone transaction to read/process/write quickly
    - Debit $90
- Isolation guarantees that phone update is not lost
  - This is a lost update

time ⇩

```
atomic{
  Read AcctA (150)

  Decrement AcctA
     by 100
  Write AcctA (50)
```

```
atomic{
  AcctA -= 90
}
```


- AcctA == 200 initially. After these two concurrent transactions AcctA==350. What property does that violate?
  - A. No property is violated
  - B. Atomicity
  - C. Isolation
  - D. Durability

```
atomic{
  AcctA += 150



}
```

```
atomic{
  AcctA -= 90
}
```

Atomicity and Isolation

- Atomicity is hard because
  - Programs make many small changes.
    - Most operations are not atomic, like x++;
  - System must be able to restore state at start of atomic operation
    - What about actions like dispensing money or firing missles?
- Isolation is hard because
  - More concurrency == more performance
  - …but system must disallow certain interleavings
  - System usually does not allow visibility of isolated state (hence the term isolated)
  - Data structures have multiple invariants that dictate constraints on a consistent update
- Mutual exclusion provides isolation
  - Most popular parallel programming technique

Parallel programming: how to provide isolation (and possibly atomicity)

Concrete Syntax for Transactions

- The concrete syntax of JDASTM.

```
Transaction tx = new Transaction(id);
boolean done = false;
while(!done) {
   try {
      tx.BeginTransaction();
      // party on my data structure!
      done = tx.CommitTransaction();
   } catch(AbortException e) {
      tx.AbortTransaction();
      done = false;
   }
}
```

Transaction's System Bookkeeping

- Transaction A's read set is R_A
  - Set of objects (addresses) read by transaction A
- Transaction B's write set is W_B
  - Set of objects (addresses) written by transaction B
- Transaction A's address set is R_A UNION W_A
  - Set of objects (addresses) read or written by transaction A

```
atomic {
  acctA -= 100;
  acctB = acctA;
}
```

Read: acctA
Write: acctA, acctB

Transactional Safety

- Conflict serializability – If one transaction writes data read or written by another transaction, then abort one transaction.
- Recoverability – No transaction that has read data from an uncommitted transaction may commit.

```
atomic {
  x++;
}
```

```
atomic {
  load t0, [x]
  add t0, 1
  store t0, [x]
}
```

- *Safe if abort transaction A or B whenever*
  - $W_A \cap (R_B \text{ UNION } W_B) \neq$ EMPTYSET

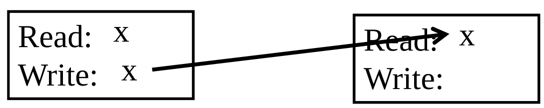
Safety examples

Transaction 0

```
atomic {
   load t0, [x]
   add t0, 1
   store t0, [x]
}
```

Transaction 1

```
atomic {
   load t0, [x]
   add t0, 1
```



Conflict: Transaction 1 should restart

How Isolation Could Be Violated

- Dirty reads
- Non-repeatable reads
- Lost updates

Restarting + I/O = Confusion

- Transactions can restart!
  - What kind of output should I expect?

```
Transaction tx = new Transaction(id);
boolean done = false;
while(!done) {
   try {
      tx.BeginTransaction();
      …
      System.out.println("Deja vu all over again");
      done = tx.CommitTransaction();
   } catch(AbortException e) {
      tx.AbortTransaction();
      done = false;
   }
}
```


Reading Uncommitted State

- What about transactional data read outside a transaction?
  - Hardware support: strong isolation for all reads
  - Software: Uncommitted state is visible
- In your lab, a lane can go from colored to white when a transaction rolls back
  - The GUI updating thread reads uncommitted state outside of a transaction
- Why would we want to read data outside of a transaction?
  - Performance

Transactional Communication

- Conflict serializability is good for keeping transactions out of each other's address sets
- Sometimes transactions must communicate
  - One transaction produces a memory value
  - Other transaction consumes the memory value
- Communication is easy to do with busy waiting
  - Just read the variable that will change
  - Transaction will restart when its written by other thread

Communicating Transactions

```
Class CokeMachine{
    …
    int count = 0;
}
```

```
CokeMachine::Deposit(){
   atomic {
     while (count == n)  ;
     Add coke to the machine;
     count++;
   }
}
```

```
CokeMachine::Remove(){
   atomic {
     while (count == 0) ;
      Remove coke from machine;
      count--;
   }
}
```

- Transactions busy-wait for each other.
  - The variable count is in the read set, so any write to count will restart the transaction

Tx Communication Without Busy-Waiting

- Retry: how to block with transactions
  - Pause transaction
  - deschedule this thread
  - Reschedule whenever another transaction conflicts with this transaction
- Transactional thread is suspended until another thread modifies data it read
  - E.g., count variable

Retry: Communication Without Busy-Wait

```
Class CokeMachine{
   …
   int count = 0;
}
```

```
CokeMachine::Deposit(){
   atomic {
     if(count == n) {retry; }
     Add coke to the machine;
     count++;
   }
}
```

```
CokeMachine::Remove(){
   atomic {
     if(count == 0) { retry; }
      Remove coke from machine;
      count--;
   }
}
```

- Scheduler and runtime cooperate to monitor address sets of transactions that are descheduled

Comparing Transactions and Monitors

```
CokeMachine::Deposit(){
   atomic {
     if(count == n) {retry; }
     Add coke to the machine;
     count++;
   }
}
```

```
CokeMachine::Remove(){
   atomic {
     if(count == 0) {retry; }
     Remove coke from machine;
     count--;
   }
}
```

```
CokeMachine::Deposit(){
   lock→acquire();
   while (count == n) {
         notFull.wait(&lock); }
   Add coke to the machine;
   count++;
   notEmpty.notify();
   lock→release();
}
```

```
CokeMachine::Remove(){
   lock→acquire();
   while (count == 0) {
         notEmpty.wait(&lock); }
   Remove coke from to the machine;
   count--;
   notFull.notify();
   lock→release();
}
```

Which is better?
A. Transactions
B. Monitors

Load linked/Store Conditional

- Load linked/store conditional.
  - Idea is to let user load a data item, compute, then store back and if "no one else" (i.e., another processor or an I/O device) has touched that memory location, then allow the store since the read-modify-write was atomic.

```
tmp = r1 = [addr];                 // Load linked into r1
do_whatever (some restrictions);
                                   // Store conditional from r2
if(tmp == [addr]) then [addr] = r2; r2 = 1;
  else r2 = 0;
```

  - Restrictions on compute: no memory accesses, limited number of instructions, no interrupts or exceptions.
- Hardware queue locks

Load linked/Store Conditional

- All of these events, if they happen between the load linked and the store conditional will cause the store conditional to fail. EXCEPT which?
  - A. Breakpoint instruction
  - B. Branch instruction
  - C. External write to loaded memory address
  - D. Return from exception instruction