Concurrency

Saman Amarasinghe
Spring 2008
ATM-single

DEMO 1

© Saman Amarasinghe 2008
public class Account {
    String id;
    String password;
    int balance;

    Account(String id, String password, 
            int balance) {
        this.id = id;
        this.password = password;
        this.balance = balance;
    }

    boolean is_password(String password) {
        return password.equals(this.password);
    }

    int getbal() {
        return balance;
    }

    void setbal(int v) {
        balance = v;
    }
}
public class Bank {
    HashMap<String, Account> accounts;
    static Bank theBank = null;

    private Bank() {
        accounts = new HashMap<String, Account>();
        add("allyssa", "mitrocks", 10000);
        add("ben", "6005iscool", 100);
    }

    public static Bank getbank() {
        if (theBank == null)
            theBank = new Bank();
        return theBank;
    }

    public Account get(String ID) {
        return accounts.get(ID);
    }

    public void add(String ID, String pass, int bal) {
        accounts.put(ID, new Account(ID, pass, bal));
    }
}
public class ATM {
    enum State { GetAccount, GetPassword, GetAction, CashAction, TransferAction };

    static Bank bnk;
    PrintWriter out;
    BufferedReader in;
    State state;

    ATM(PrintWriter out, BufferedReader in) {
        this.out = out;
        this.in = in;
        this.state = State.GetAccount;
    }

    public static void main(String[] args) throws IOException {
        bnk = Bank.getbank();

        ServerSocket ss = null;
        try {
            ss = new ServerSocket(4444);
        } catch (IOException e) {
            System.err.println("Problem port: 4444.");
            System.exit(1);
        }

        PrintWriter out = new PrintWriter(ss.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new InputStreamReader(ss.getInputStream()));

        ATM atm = new ATM(out, in);
        atm.run();
    }
}
A Bank ATM Machine

Invalid account → Get Account

Valid account → Get password

Invalid password → Quit

Valid password → Get Action

Invalid Input → Get Action

Done Cash → Cash Action

Cash → Transfer

Transfer → Transfer Action

Done Xfer → Transfer Action

© Saman Amarasinghe 2008
A Bank ATM Machine

```java
public void run() {
    Account acc = null;
    while(true) {
        try {
            switch(state) {
                case GetAccount:
                    out.println("WELCOME TO DJ&SA BANK");
                    out.println(" ");
                    out.println("Account ID > ");
                    String id = in.readLine();
                    acc = bnk.get(id);
                    if (acc == null)
                        state = State.GetAccount;
                    else
                        state = State.GetPassword;
                    break;

                case GetPassword:
                    out.println("Password > ");
                    String pass = in.readLine();
                    if (!acc.is_password(pass))
                        state = State.GetAccount;
                    else
                        state = State.GetAction;
                    break;

                case CashAction:
                    CashAction(acc);
                    state = State.GetAction;
                    break;

                case TransferAction:
                    TransferAction(acc);
                    state = State.GetAction;
                    break;

                case GetAction:
                    out.println("'c'cash, 't'transfer, 'q'quit > ");
                    String action = in.readLine();
                    switch(action.charAt(0)) {
                        case 'c':
                            state = State.CashAction;
                            break;
                        case 't':
                            state = State.TransferAction;
                            break;
                        case 'q':
                            out.println("balance is $" + acc.getbal());
                            state = State.GetAccount;
                            break;
                        default:
                            state = State.GetAccount;
                            break;
                    }
                    break;
            }
        } catch(Exception e) {
            state = State.GetAccount;
        }
    }
}
```

void CashAction (Account acc) throws IOException {
    out.println("balance is "+acc.getbal());
    out.println("Deposit or withdraw amount > ");
    int val = Integer.valueOf(in.readLine());
    int curr = acc.getbal();
    if (curr + val < 0) {
        out.println("Insufficient Balance");
        return;
    }
    curr = curr + val;
    acc.setbal(curr);
    out.println("New balance is "+curr);
}
A Very Busy ATM? What to do?

Too many people at the ATM?

Get a second ATM

Can we run two copies of the Bank software?

Need concurrency!!
What is concurrency?

What is a sequential program?
➢ A single thread of control that executes one instruction and when it is finished execute the next logical instruction

What is a concurrent program?
➢ A collection of autonomous sequential threads, executing (logically) in parallel

The implementation (i.e. execution) of a collection of threads can be:
Multiprogramming
   • Threads multiplex their executions on a single processor.
Multiprocessing
   • Threads multiplex their executions on a multiprocessor or a multicore system
Distributed Processing
   • Processes multiplex their executions on several different machines
Concurrency and Parallelism

Concurrency is not (only) parallelism

**Interleaved Concurrency**
- Logically simultaneous processing
- Interleaved execution on a single processor

**Parallelism**
- Physically simultaneous processing
- Requires a multiprocessors or a multicore system
Java has a predefined class `java.lang.Thread` which provides the mechanism by which threads are created

```java
public class MyThread extends Thread {
    public void run() {
    }
}
```

However to avoid all threads having to be subtypes of `Thread`, Java also provides a standard interface

```java
public interface Runnable {
    public void run();
}
```

Hence, any class which wishes to express concurrent execution must implement this interface and provide the `run` method

Threads do not begin their execution until the `start` method in the `Thread` class is called
**Why use Concurrent Programming?**

**Natural Application Structure**
- The world is not sequential! Easier to program multiple independent and concurrent activities.

**Increased application throughput and responsiveness**
- Not blocking the entire application due to blocking IO

**Performance from multiprocessor/multicore hardware**
- Parallel execution

**Distributed systems**
- Single application on multiple machines
- Client/server type or peer-to-peer systems
Multiple ATM Machines

```java
public class ATM extends Thread {
    enum State { GetAccount, GetPassword, GetAction, CashAction, TransferAction }

    static Bank bnk;
    static final int numATMs = 2;

    PrintWriter out;
    BufferedReader in;
    State state;

    ATM(PrintWriter out, BufferedReader in) {
        this.out = out;
        this.in = in;
        this.state = State.GetAccount;
    }

    public static void main(String[] args) throws IOException {
        bnk = Bank.getBank();
        ATM atm[] = new ATM[numATMs];

        ServerSocket sS = null;
        try {
            sS = new ServerSocket(4444);
        } catch (IOException e) {
            System.err.println("Problem port: 4444.");
            System.exit(1);
        }

        for(int i=0; i< numATMs; i++) {
            Socket cS = null;
            try {
                cS = sS.accept();
            } catch (IOException e) {
                System.err.println("Accept failed.");
                System.exit(1);
            }

            PrintWriter out = new PrintWriter(cS.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(cS.getInputStream()));

            Atm[i]= new ATM(out, in);
            Atm[i].start();
        }
    }
}
© Saman Amarasinghe 2008
```
ATM-multi-cash-nosync

DEMO II

© Saman Amarasinghe 2008
```java
void CashAction (Account acc) throws IOException {
    out.println("balance is "+ acc.getbal());
    out.println("Deposit or withdraw amount > ");
    int val = Integer.valueOf(in.readLine());

    int curr = acc.getbal();
    if (curr + val < 0) {
        out.println("Insufficient Balance");
        return;
    }

    curr = curr + val;
    acc.setbal(curr);
    out.println("balance "+ curr);
}
```
Races

Race conditions – insidious bugs
- Non-deterministic, timing dependent
- Cause data corruption, crashes
- Difficult to detect, reproduce, eliminate

Many programs contain races
- Inadvertent programming errors
- Failure to observe locking discipline
Data Races

A data race happens when two threads access a variable simultaneously, and one access is a write.

```c
int t1;
t1 = hits;
hits = t1 + 1;
int t2;
t2 = hits;
hits = t2 + 1;
```
Data Races

A data race happens when two threads access a variable simultaneously, and one access is a write

```c
int t1;
t1 = hits;
hits = t1+1;

int t2;
t2 = hits;
hits = t2+1;
```
A data race happens when two threads access a variable simultaneously, and one access is a write.

```c
int t1;
t1 = hits;
hits = t1 + 1;
```

```c
int t2;
t2 = hits;
hits = t2 + 1;
```
Data Races

Problem with data races: non-determinism
- Depends on interleaving of threads

Usual way to avoid data races:
- Serialized access of all the shared objects
Synchronization

All the interleavings of the threads are NOT acceptable correct programs.

Java provides synchronization mechanism to restrict the interleavings

Synchronization serves two purposes:

- **Ensure safety** for shared updates
  - Avoid race conditions
- **Coordinate** actions of threads
  - Parallel computation
  - Event notification
Multiple threads access shared resource simultaneously

Safe only if:

- All accesses have no effect on resource,
  - e.g., reading a variable,
- or
- All accesses idempotent
  - E.g., \( y = \text{sign}(a), a = a*2; \)
- or
- Only one access at a time:
  *mutual exclusion*
Safety: Example

“*The too much milk problem*”

<table>
<thead>
<tr>
<th>time</th>
<th>You</th>
<th>Your Roommate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Arrive home</td>
<td>Arrive home</td>
</tr>
<tr>
<td>3:05</td>
<td>Look in fridge, no milk</td>
<td>Look in fridge, no milk</td>
</tr>
<tr>
<td>3:10</td>
<td>Leave for grocery</td>
<td>Leave for grocery</td>
</tr>
<tr>
<td>3:15</td>
<td>Arrive at grocery</td>
<td>Buy Milk</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive at grocery</td>
<td></td>
</tr>
<tr>
<td>3:25</td>
<td>Buy milk</td>
<td>Buy Milk</td>
</tr>
<tr>
<td>3:35</td>
<td>Arrive home, put milk in fridge</td>
<td>Arrive home, put up milk</td>
</tr>
<tr>
<td>3:45</td>
<td></td>
<td>Oh no!</td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Model of need to synchronize activities

Courtesy of Emery Berger @ UMASS
Mutual Exclusion

Prevent more than one thread from accessing *critical section* at a given time

- Once a thread is in the critical section, no other thread can enter that critical section until the first thread has left the critical section.
- No interleavings of threads within the critical section
- *Serializes* access to section

```java
synchronized int getbal() {
    return balance;
}

synchronized void post(int v) {
    balance = balance + v;
}
```
Atomicity

Synchronized methods execute the body as an atomic unit.

May need to execute a code region as the atomic unit.

Block Synchronization is a mechanism where a region of code can be labeled as synchronized.

The `synchronized` keyword takes as a parameter an object whose lock the system needs to obtain before it can continue.

Example:

```java
synchronized (acc) {
    curr = acc.getbal();
    curr = curr + val;
    acc.setbal(curr);
}
```
void CashAction (Account acc) throws IOException {

    out.println("balance is $" + acc.getbal());

    out.println("Deposit or withdraw amount > ");
    int val = Integer.valueOf(in.readLine());

    synchronized (acc) {
        int curr = acc.getbal();
        if (curr + val < 0) {
            out.println("Insufficient Balance");
            return;
        }

        curr = curr + val;

        acc.setbal(curr);
        out.println("balance $" + curr);
    }
}
ATM-multi-cash-get-and-set-sync

DEMO III

© Saman Amarasinghe 2008
void CashAction (Account acc) throws IOException {

    out.println("balance is $" + acc.getbal());

    out.println("Deposit or withdraw amount > ");
    int val = Integer.valueOf(in.readLine());

    synchronized (acc) {
        int curr = acc.getbal();
        if (curr + val < 0) {
            out.println("Insufficient Balance");
            return;
        }

        curr = curr + val;

        acc.setbal(curr);
        out.println("balance $" + curr);
    }
}
public class Account {
    String id;
    String password;
    int balance;

    Account(String id, String password, int balance) {
        this.id = id;
        this.password = password;
        this.balance = balance;
    }

    boolean is_password(String password) {
        return password.equals(this.password);
    }

    int getbal() {
        return balance;
    }

    void setbal(int v) {
        balance = v;
    }

    synchronized int post(int v) {
        balance = balance + v;
        return balance;
    }
}

void CashAction (Account acc) throws IOException {
    out.println("balance is \$" + acc.getbal());

    out.println("Deposit or withdraw amount > ");
    int val = Integer.valueOf(in.readLine());

    int curr = acc.getbal();
    if (curr + val < 0) {
        out.println("Insufficient Balance");
        return;
    }

    int newbal = acc.post(val);
    out.println("balance \$" + newbal);
}
DEMO IV
Let the Account be Atomic

void CashAction (Account acc) throws IOException {

    out.println("balance is $" + acc.getbal());

    out.println("Deposit or withdraw amount > ");
    int val = Integer.valueOf(in.readLine());

    int curr = acc.getbal();
    if (curr + val < 0) {
        out.println("Insufficient Balance");
        return;
    }

    int newbal = acc.post(val);
    out.println("balance $" + newbal);
}
void TransferAction (Account fracc) throws Exception {
    out.println("Destination Account ID > ");
    String id = in.readLine(); System.out.println(id);
    Account toacc = bnk.get(id);
    if (toacc == null)
        return;

    out.println("your balance is $" + fracc.getbal());

    out.println("Transfer amount > ");
    int val = Integer.valueOf(in.readLine());
    if(val < 0) {
        out.println("Can’t withdraw from other accounts");
        return;
    }

    synchronized(fracc) {
        int frcurr = fracc.getbal();
        if (frcurr - val < 0) {
            out.println("Insufficient Balance");
            return;
        }
    }

    synchronized(toacc) {
        int tocurr = toacc.getbal();

        frcurr = frcurr - val;
        tocurr = tocurr + val;

        toacc.setbal(tocurr);
    }

    fracc.setbal(frcurr);
    out.println("New balance is $" + frcurr);
}
ATM-multi-xfer-deadlock

DEMO V

© Saman Amarasinghe 2008
State Machine

TransferAction

Transfer Amount

Get From Balance

Get To Balance

Calculate

Set To Balance

Set From Balance

End CashAction

TransferAction

X

Ta

Gf

Gt

Ca

St

Sf

Ea

Y

Ta

Gf

Gt

Ca

St

Sf

Ea

P

Q

NP

NQ

YP

YQ

XP

XQ

uXP

uXQ

uYP

uYQ

Insufficient funds

Negative transfer
Avoiding Deadlock

Cycle in locking graph = deadlock

Standard solution: canonical order for locks
- Acquire in increasing order
- Release in decreasing order

Ensures deadlock-freedom, but not always easy to do
There are 5 philosophers sitting at a round table.
Between each adjacent pair of philosophers is a chopstick.
Each philosopher does two things: think and eat.

- The philosopher thinks for a while.
- When the philosopher becomes hungry, she stops thinking and...
  - Picks up left and right chopstick
  - He cannot eat until he has both chopsticks, has to wait until both chopsticks are available
  - When the philosopher gets the two chopsticks she eats
- When the philosopher is done eating he puts down the chopsticks and begins thinking again.
import java.io.*;
import java.util.*;

public class Philosopher extends Thread {
    static final int count = 5;
    Chopstick left;
    Chopstick right;
    int position;

    Philosopher(int position, Chopstick left, Chopstick right) {
        this.position = position;
        this.left = left;
        this.right = right;
    }

    public static void main(String[] args) {
        Philosopher phil[] = new Philosopher[count];

        Chopstick last = new Chopstick();
        Chopstick left = last;
        for(int i=0; i<count; i++) {
            Chopstick right = (i==count-1) ? last : new Chopstick();

            phil[i] = new Philosopher(i, left, right);
            left = right;
        }

        for(int i=0; i<count; i++) {
            phil[i].start();
        }
    }

    ...
Dining Philosophers Problem: Take I

```java
public void run() {
    try {
        while(true) {
            synchronized(left) {
                synchronized(right) {
                    System.out.println(times + ": Philosopher " + position + " is done eating");
                }
            }
        }
    } catch (Exception e) {
        System.out.println("Philosopher " + position + "'s meal got disturbed");
    }
}
```
Dining Philosophers Problem: Take II

```java
static Object table;
public void run() {
    try {
        while(true) {
            synchronized(table) {
                synchronized(left) {
                    synchronized(right) {
                        System.out.println(times + " : Philosopher " + position + " is done eating");
                    }
                }
            }
        }
    } catch (Exception e) {
        System.out.println("Philosopher " + position + ", meal got disturbed");
    }
}
```
public void run() {
    try {
        Chopstick first = (position%2 == 0)?left:right;
        Chopstick second = (position%2 == 0)?right:left;
        while(true) {
            synchronized(first) {
                synchronized(second) {
                    System.out.println(times + " : Philosopher " + position + " is done eating");
                }
            }
        }
    } catch (Exception e) {
        System.out.println("Philosopher " + position + ",'s meal got disturbed");
    }
}
Other types of Synchronization

There are a lot of ways to use Concurrency in Java

- Semaphores
- Blocking & non-blocking queues
- Concurrent hash maps
- Copy-on-write arrays
- Exchangers
- Barriers
- Futures
- Thread pool support
Potential Concurrency Problems

**Deadlock**
- Two or more threads stop and wait for each other

**Livelock**
- Two or more threads continue to execute, but make no progress toward the ultimate goal.

**Starvation**
- Some thread gets deferred forever.

**Lack of fairness**
- Each thread gets a turn to make progress.

**Race Condition**
- Some possible interleaving of threads results in an undesired computation result.
Conclusion

Concurrency and Parallelism are important concepts in Computer Science

Concurrency can simplify programming

- However it can be very hard to understand and debug concurrent programs

Parallelism is critical for high performance

- From Supercomputers in national labs to Multicores and GPUs on your desktop
Lecture Exercises

Draw the state machine for synchronization using `post()` in the account class.

Draw the product machine and show how it has an undesirable path that leads to a negative balance.

Propose a solution that overcomes this problem.

In the current synchronized `CashAction`, you may get an insufficient funds message even when you withdraw an amount smaller than the balance displayed before withdrawal. How is this possible?

Propose a solution to this problem.

Is this solution robust? (Think of the behavior of your Bank’s ATM. Does it have the above behavior?)

Propose a deadlock free implementation for `CashTransfer`.

Is this a fair solution?

Give an example of a livelock.