6.189 IAP 2007

Lecture 4

Concurrent Programming

In this lecture…

- Study concurrent programming with an emphasis on correctness
  - Parallel programs have the same correctness issues
- Start with a simpler and easier machine/programming model
  - Use Java as a language
  - Use an Abstract Shared Memory Machine Model
- Next Lecture..
  - Use C/C++ primitives (MPI)
  - Study parallel programming with an emphasis on performance
  - Using a distributed memory machine

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:

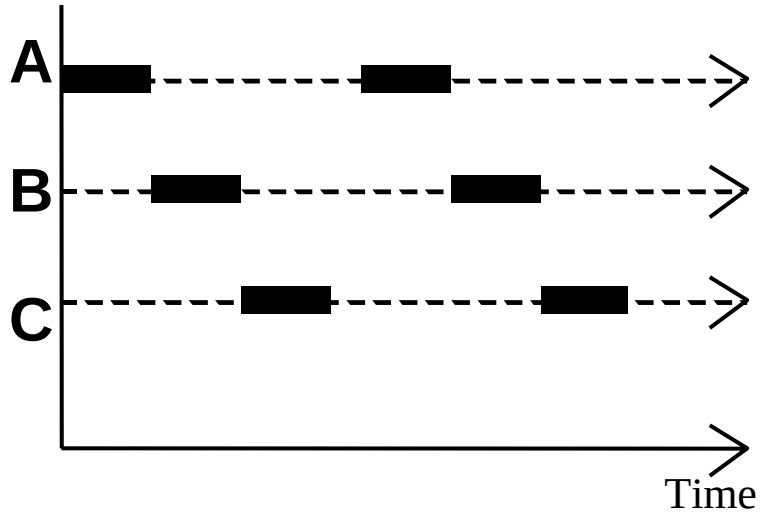
  Multiprogramming
  - Threads multiplex their executions on a single processor.

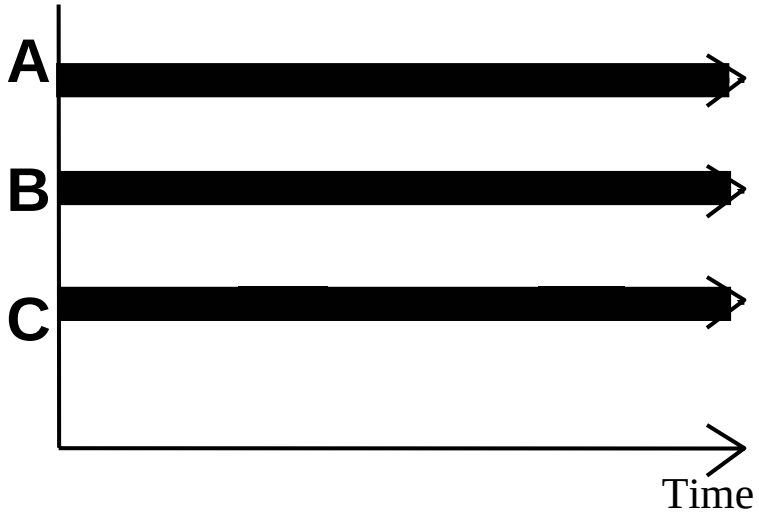
  Multiprocessing
  - Threads multiplex their executions on a multiprocessor or a multicore system

  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





Account and Bank

```
import java.util.*;

public class Account {
      String id;
      String password;
      int balance;

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

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

      int getbal() {
            return balance;
      }

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

}
```

```
import java.util.*;

public class Bank {
      HashMap<String, Account> accounts;
      static Bank theBank = null;

      private Bank() {
            accounts = new HashMap<String, Account>();
      }

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

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

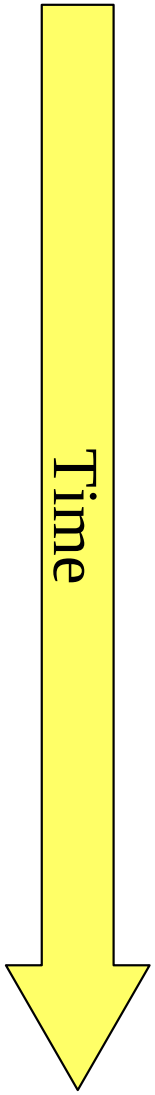
ATM

```
import java.util.*;
import java.io.*;

public class ATM {
      static Bank bnk;
      PrintStream out;
      BufferedReader in;

      ATM(PrintStream out, BufferedReader in) {
         this.out = out;
         this.in = in;
      }

       public static void main(String[] args) {
          bnk = Bank.getbank();
         BufferedReader stdin = new BufferedReader
              (new InputStreamReader(System.in));
         ATM atm = new ATM(System.out, stdin);
         atm.run();
       }
```

```
public void run() {
      while(true) {
         try {
            out.print("Account ID > " );
            String id = in.readLine();
            String acc = bnk.get(id);
            if (acc == null) throw new Exception();
            out.print("Password > " );
            String pass = in.readLine();
            if (!acc.is_password(pass))
               throw new Exception();
            out.print("your balance is "  +  acc.getbal());
            out.print("Deposit or withdraw amount > " );
            int val = in.read();
            if (acc.getbal() + val > 0)
                acc.post(val);
           else
               throw new Exception();
            out.print("your balance is "  +  acc.getbal());
         } catch(Exception e) {
            out.println("Invalid input, restart" );
         }
      }
   }
}
```

Activity trace

ATM



ATM

```
import java.util.*;
import java.io.*;

public class ATM {
      static Bank bnk;
      PrintStream out;
      BufferedReader in;

      ATM(PrintStream out, BufferedReader in) {
            this.out = out;
            this.in = in;
      }

       public static void main(String[] args) {
            bnk = Bank.getbank();
            BufferedReader stdin = new BufferedReader
                 (new InputStreamReader(System.in));
            ATM atm = new ATM(System.out, stdin);
            atm.run();
       }
```

```
public void run() {
      while(true) {
          try {
              out.print("Account ID > " );
              String id = in.readLine();
              String acc = bnk.get(id);
              if (acc == null) throw new Exception();
              out.print("Password > " );
              String pass = in.readLine();
              if (!acc.is_password(pass))
                 throw new Exception();
              out.print("your balance is "  +  acc.getbal());
              out.print("Deposit or withdraw amount > "
    );
              int val = in.read();
              if (acc.getbal() + val > 0)
                  acc.post(val);
              else
                 throw new Exception();
              out.print("your balance is "  +  acc.getbal());
          } catch(Exception e) {
              out.println("Invalid input, restart" );
          }
      }
  }
}
```

I need to run multiple ATM machines from my program, how do I do that?

Concurrency in Java

- Java has a predefined class `java.lang.Thread` which provides the mechanism by which threads are created

```
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

```
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 ATMs

```
import java.util.*;
import java.io.*;

public class ATM {

    static Bank bnk;
    PrintStream out;
    BufferedReader in;


    ATM(PrintStream out, BufferedReader in) {
        this.out = out;
        this.in = in;
    }


     public static void main(String[] args) {
         bnk = Bank.getbank();
         BufferedReader stdin = new BufferedReader
             (new InputStreamReader(System.in));
         ATM atm = new ATM(System.out, stdin);
         atm.run();
     }
```

```
public void run() {
        while(true) {
            try {
                out.print("Account ID > " );
                String id = in.readLine();
                String acc = bnk.get(id);
                if (acc == null) throw new Exception();
                out.print("Password > " );
                String pass = in.readLine();
                if (!acc.is_password(pass))
                   throw new Exception();
                out.print("your balance is "  +  acc.getbal());
                out.print("Deposit or withdraw amount > " );
                int val = in.read();
                if (acc.getbal() + val > 0)
                    acc.post(val);
               else
                   throw new Exception();
                out.print("your balance is "  +  acc.getbal());
            } catch(Exception e) {
                out.println("Invalid input, restart" );
            }
        }
    }
}
```

I need to run multiple ATM machines from my program, how do I do that?

Multiple ATMs

```
import java.util.*;
import java.io.*;

public class ATMs extends Thread {
      static final int numATMs = 4;
      static Bank bnk;
      PrintStream out;
      BufferedReader in;
      int atmnum;

      ATMs(int num, PrintStream out, BufferedReader in) {
            this.out = out;
            this.in = in;
            this.atmnum = num;
      }

       public static void main(String[] args) {
            bnk = Bank.getbank();
            ATMs atm[] = new ATMs[numATMs];
            for(int i=0; i<numATMs; i++){
                atm[i] = new ATMs(i, outdevice(i), indevice(i));
                atm[i].start();
            }
       }
```

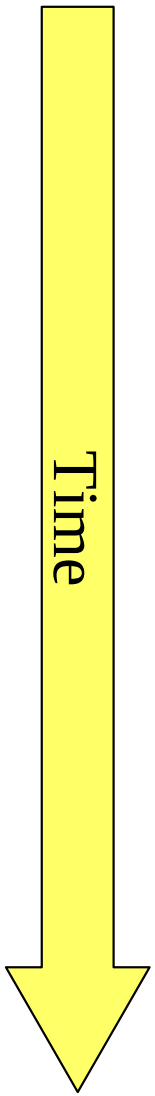
```
public void run() {
      while(true) {
          try {
              out.print("Account ID > " );
              String id = in.readLine();
              String acc = bnk.get(id);
              if (acc == null) throw new Exception();
              out.print("Password > " );
              String pass = in.readLine();
              if (!acc.is_password(pass))
                 throw new Exception();
              out.print("your balance is "  +  acc.getbal());
              out.print("Deposit or withdraw amount > " );
              int val = in.read();
              if (acc.getbal() + val > 0)
                  acc.post(val);
             else
                 throw new Exception();
              out.print("your balance is "  +  acc.getbal());
          } catch(Exception e) {
              out.println("Invalid input, restart" );
          }
      }
  }
}
```

I need to run multiple ATM machines from my program, how do I do that?

Activity trace

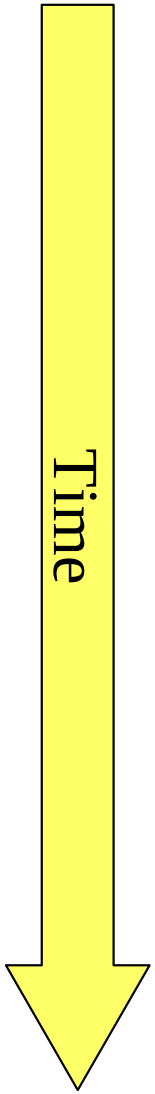
ATM 1

ATM 2



Activity trace II

ATM 1 **ATM 2**



100 - 90 - 90 = 10!!!

Activity trace II

balance

100

100

100

10

10

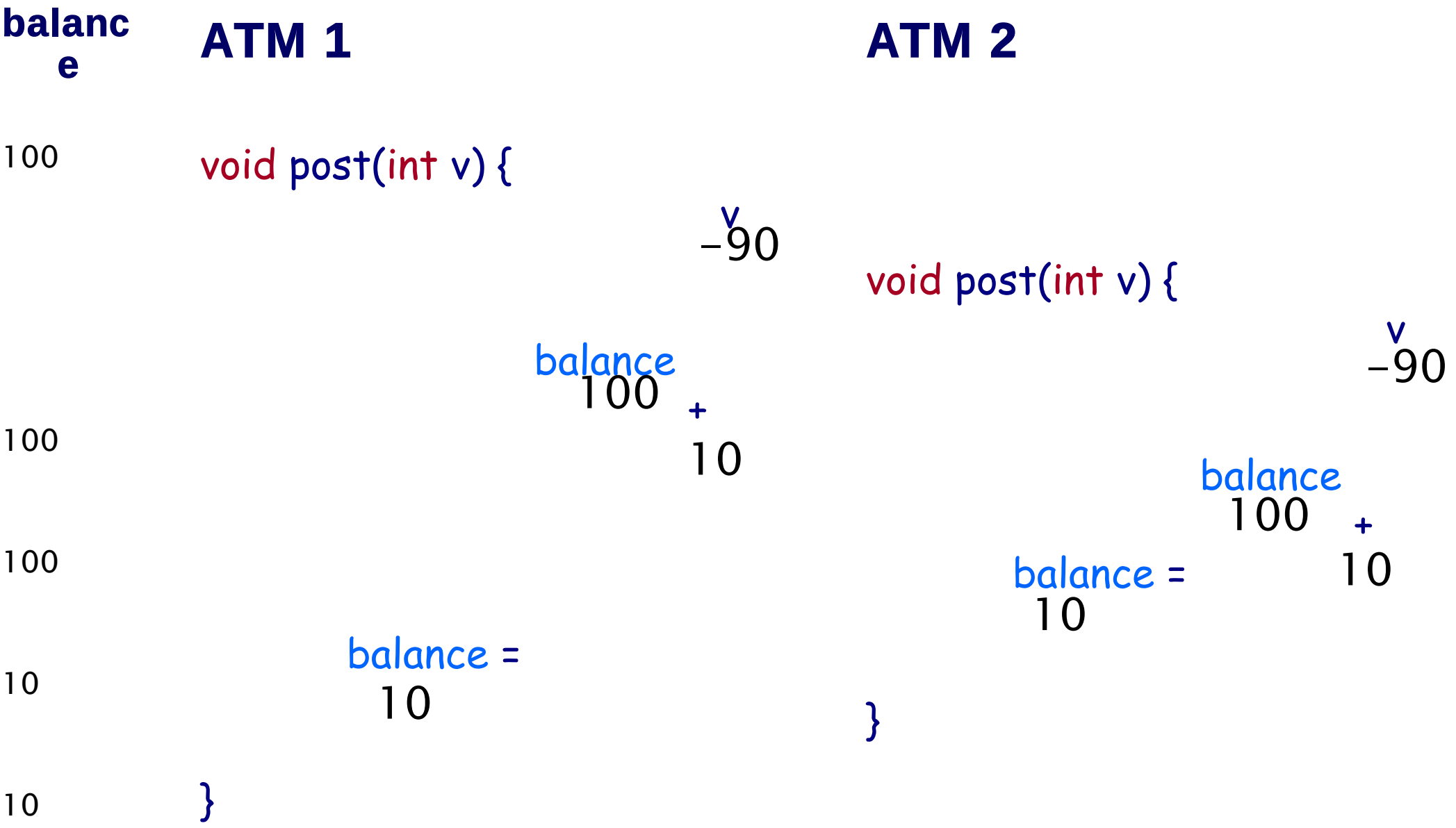
ATM 1

```
out.print("your balance is " + acc.getbal());
Your account balance is 100

out.print("Deposit or withdraw amount > ");
Deposit or Withdraw amount >

     -90
int val = in.read();

if (acc.getbal() + val > 0)


 acc.post(val);


out.print("your balance is " + acc.getbal());
Your account balance is 10
```

ATM 2

```
out.print("your balance is " + acc.getbal());
Your account balance is 100


out.print("Deposit or withdraw amount > ");
Deposit or Withdraw amount >


     -90

int val = in.read();


if (acc.getbal() + val > 0)

  acc.post(val);



out.print("your balance is " + acc.getbal());
Your account balance is 10
```

Activity trace II



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

Safety

- 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 = sign(a)`, `a = a*2;`

  or

  - Only one access at a time: *mutual exclusion*

Safety: Example

- "The *too much milk* problem"

time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy Milk
3:50		Arrive home, put up milk
3:50		Oh no!

- Model of need to **synchronize** activities

Why You Need Locks

thread A

```
if (no milk && no note)
   leave note
   buy milk
   remove note
```

thread B

```
if (no milk && no note)
   leave note
   buy milk
   remove note
```

- Does this w

too much milk

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

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

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

Activity trace II zoomed-in

balance	ATM 1	ATM 2
100		int val = in.read();
	int val = in.read();	
100		if (acc.getbal() + val > 0)
100	if (acc.getbal() + val > 0)	
100		acc.post(val);
10	acc.post(val);	
-80		
		out.print("*your balance is* " + acc.getbal());
		Your account balance is -80
	out.print("*your balance is* " + acc.getbal());	
	Your account balance is -80	

Negative Bank Balance!

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:

```
synchronized (acc) {
        if (acc.getbal() + val > 0)
                acc.post(val);
        else
                throw new Exception();
        out.print("your balance is " + acc.getbal());
}
```

Synchronizing a block

```
import java.util.*;
import java.io.*;

public class ATMs extends Thread {
      static final int numATMs = 1;
      static Bank bnk;
      PrintStream out;
      BufferedReader in;
      int atmnum;

      ATMs(int num, PrintStream out, BufferedReader in) {
            this.out = out;
            this.in = in;
            this.atmnum = num;
      }

       public static void main(String[] args) {
            bnk = Bank.getbank();
            ATMs atm[] = new ATMs[numATMs];
            for(int i=0; i<numATMs; i++){
                atm[i] = new ATMs(i, outdevice(i), indevice(i));
                atm[i].start();
            }
       }
```

```
public void run() {
      while(true) {
          try {
              out.print("Account ID > " );
              String id = in.readLine();
              String acc = bnk.get(id);
              if (acc == null) throw new Exception();
              out.print("Password > " );
              String pass = in.readLine();
              if (!acc.is_password(pass))
                 throw new Exception();
              out.print("your balance is "  +  acc.getbal());
              out.print("Deposit or withdraw amount > " );
              int val = in.read();
              synchronized (acc) {
                 if (acc.getbal() + val > 0)
                   acc.post(val);
                 else
                   throw new Exception();
                 out.print("your balance is "  +  acc.getbal());
              }
          } catch(Exception e) {
              out.println("Invalid input, restart" );
          }
      }
  }
}
```

Activity trace II

balance

ATM 1

100

```
out.print("your balance is " + acc.getbal());
```
100

```
Your account balance is 100
```

```
out.print("Deposit or withdraw amount > ");
Deposit or Withdraw amount >
```

```
-90
int val = in.read();
```

100

```
synchronized(acc)
```
100

```
if (acc.getbal() + val > 0)
 acc.post(val);
```
10

```
out.print("your balance is " + acc.getbal());
```
10

```
Your account balance is 10
```
10

ATM 2

```
out.print("your balance is " + acc.getbal());
Your account balance is 100
```

```
out.print("Deposit or withdraw amount > ");
Deposit or Withdraw amount >
```

```
-90
int val = in.read();
```

```
synchronized(acc)
```

```
if (acc.getbal() + val > 0)
   throw new Exception()
```

Balance shows 100, but couldn't withdraw!

Synchronizing a block

```
import java.util.*;
import java.io.*;

public class ATMs extends Thread {
      static final int numATMs = 1;
      static Bank bnk;
      PrintStream out;
      BufferedReader in;
      int atmnum;

      ATMs(int num, PrintStream out, BufferedReader in) {
            this.out = out;
            this.in = in;
            this.atmnum = num;
      }

       public static void main(String[] args) {
            bnk = Bank.getbank();
            ATMs atm[] = new ATMs[numATMs];
            for(int i=0; i<numATMs; i++){
                atm[i] = new ATMs(i, outdevice(i), indevice(i));
                atm[i].start();
            }
       }
```

```
public void run() {
      while(true) {
          try {
              out.print("Account ID > " );
              String id = in.readLine();
              String acc = bnk.get(id);
              if (acc == null) throw new Exception();
              out.print("Password > " );
              String pass = in.readLine();
              if (!acc.is_password(pass))
                  throw new Exception();
              synchronized (acc) {
                  out.print("your balance is "  +  acc.getbal());
                  out.print("Deposit or withdraw amount > " );
                  int val = in.read();
                  if (acc.getbal() + val > 0)
                    acc.post(val);
                  else
                    throw new Exception();
                  out.print("your balance is "  +  acc.getbal());
              }
          } catch(Exception e) {
              out.println("Invalid input, restart" );
          }
      }
  }
}
```

Activity trace II

ATM 1

```
Account ID >
```

ben

```
Password >
```

6189cell

```
synchronized(acc)
out.print("your balance is " + acc.getbal());
Your account balance is 100

out.print("Deposit or withdraw amount > ");
Deposit or Withdraw amount >
```

-90

ATM 2

```
Account ID >
```

ben

```
Password >
```

6189cell

```
synchronized(acc)
```

Account transfers

```
public boolean transfer(Account from, Account to, int val) {
        synchronized(from) {
            if (from.getbal() > val)
                from.post(-val);
            else
                throw new Exception();
            synchronized(to) {
                to.post(val);
            }
        }
    }
```

Account Transfers

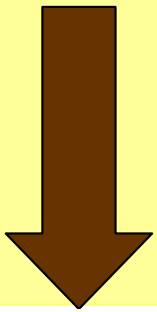
Allyssa wants to transfer $10 to Ben's account
While Ben wants to also transfer $20 to Allyssa's account

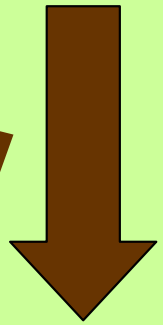
Allyssa→Ben

```
synchronized(from)
if (from.getbal() > val)
from.post(-val);

    synchronized(to)
    Waiting for Ben's account
    to be released to perform
```

Ben→Allysa

```
synchronized(from)
if (from.getbal() > val)
from.post(-val);

    synchronized(to)
    Waiting for Allyssa's account
    to be released to perform
```





DEADLOCKED!

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

Account and Bank

```
public class Account {
    String id;
    String password;
    int balance;
    static int count;

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

    …
}
```

```
…
public boolean transfer(Account from,
                        Account to,
                        int val) {

    synchronized(from) {
        synchronized(to) {
            if (from.getbal() > val)
                from.post(-val);
            else
                throw new Exception();
            to.post(val);
        }
    }
}
…
```

Account and Bank

```
public class Account {
    String id;
    String password;
    int balance;
    static int count;
    public int rank;


    Account(String id,
            String password,
            String balance) {
        this.id = id;
        this.password = password;
        this.balance = balance;
        rank = count++;
    }

    …

}
```

```
…
public boolean transfer(Account from,
                        Account to,
                        int val) {

    Account first = (from.rank > to.rank)?from:to;
    Account second = (from.rank > to.rank)?to:from;
        synchronized(first) {
            synchronized(second) {
                if (from.getbal() > val)
                    from.post(-val);
                else
                    throw new Exception();
                to.post(val);
            }
        }
}
…
```

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

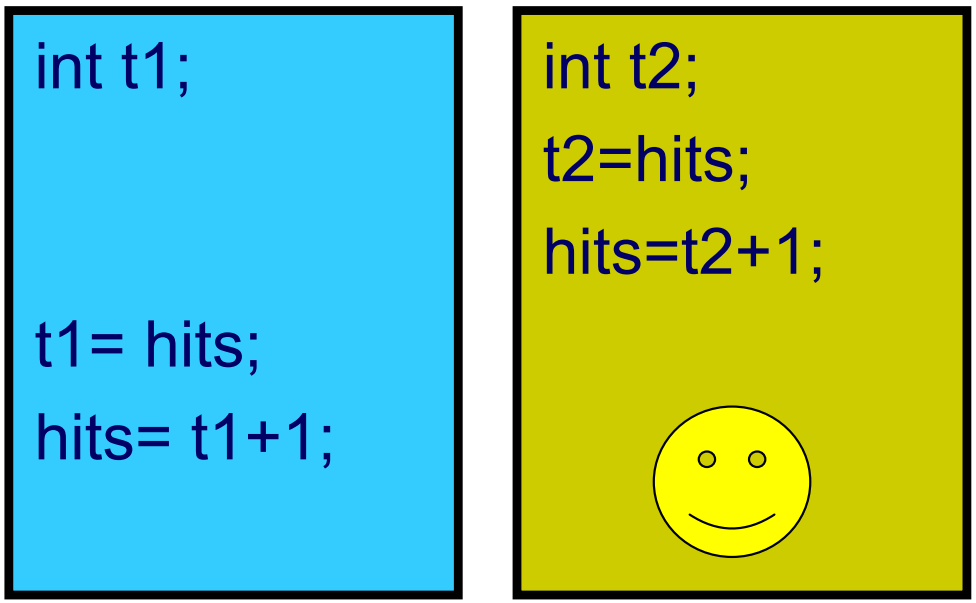
- A **data race** happens when two threads access a variable simultaneously, and one access is a *write*

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

```
int t2;
t2=hits;
hits=t2+1;
```

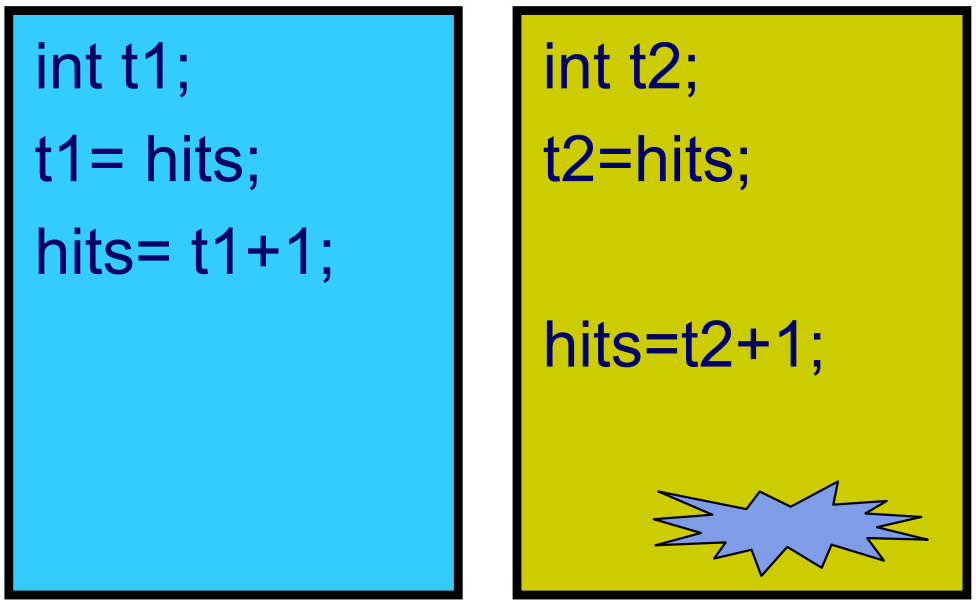
Data Races

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



Data Races

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

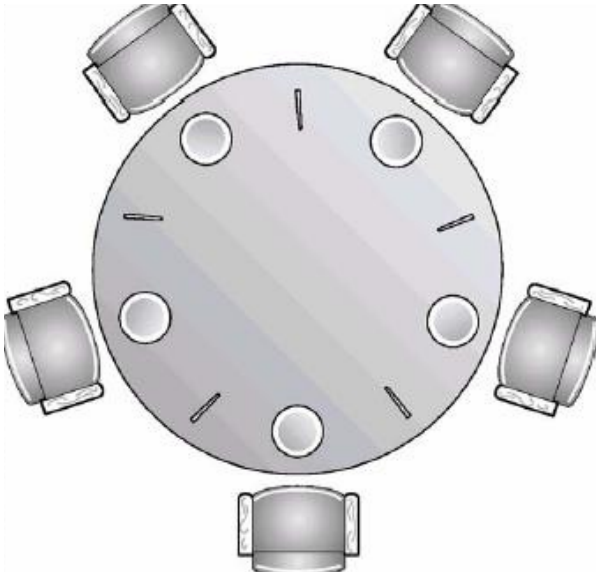


Data Races

- Problem with data races:
  non-determinism
  - Depends on interleaving of threads
- Usual way to avoid data races:
  mutual exclusion
  - Ensures **serialized** access of all the shared objects

Dining Philosophers Problem

- 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.



Dining Philosophers Problem Setup

```
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

```
public void run() {
        try {
            while(true) {
1                   synchronized(left) {
2                       synchronized(right) {
3                           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

```
static Object table;
public void run() {
        try {
           while(true) {
1                  synchronized(table) {
2                      synchronized(left) {
3                          synchronized(right) {
4                              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 III

```
public void run() {
    try {
        Chopstick first = (position%2 == 0)?left:right;
        Chopstick second = (position%2 == 0)?right:left;
            while(true) {
1                   synchronized(first) {
2                       synchronized(second) {
3                           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
- Concurrency is the basis for writing parallel programs
- Next Lecture: How to write a Parallel Program