Parallelism (PAR)

Introduction and motivation

Eduard Ayguadé, Josep Ramon Herrero and Daniel Jiménez

Computer Architecture Department
Universitat Politècnica de Catalunya

2014/15-Fall

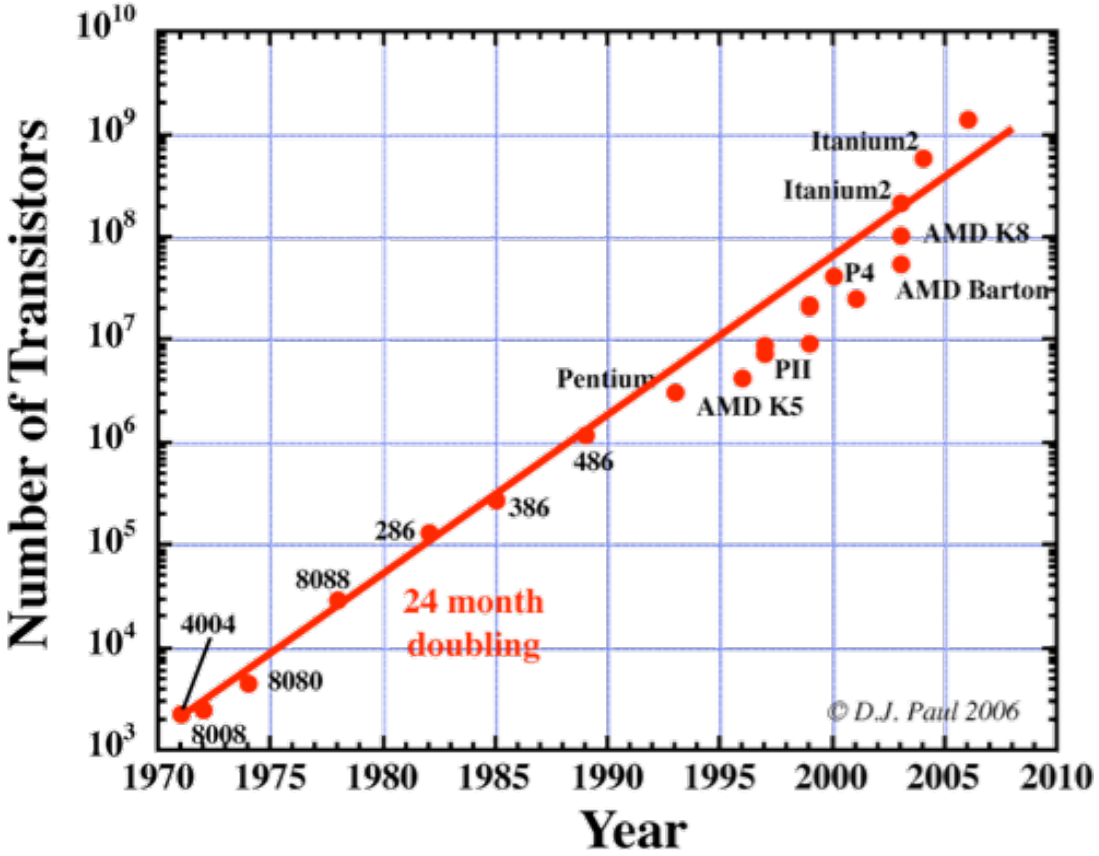
Outline

Programming evolution and Moore's law

- 60s and 70s
  - Assembly language was used
  - Computers able to handle large and complex programs
  - Need to get abstraction and portability without losing performance
  - High-level languages: FORTRAN and C
- 80s and 90s
  - Inability to build and maintain complex and robust applications requiring multi-million lines of code developed by hundreds of programmers
  - Computers could handle even larger more complex programs
  - Needed to get composability and maintainability
  - Object Oriented Programming: C++, C# and Java
  - Performance was not an issue (compilers and Moore's Law)

Moore's law

The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years.



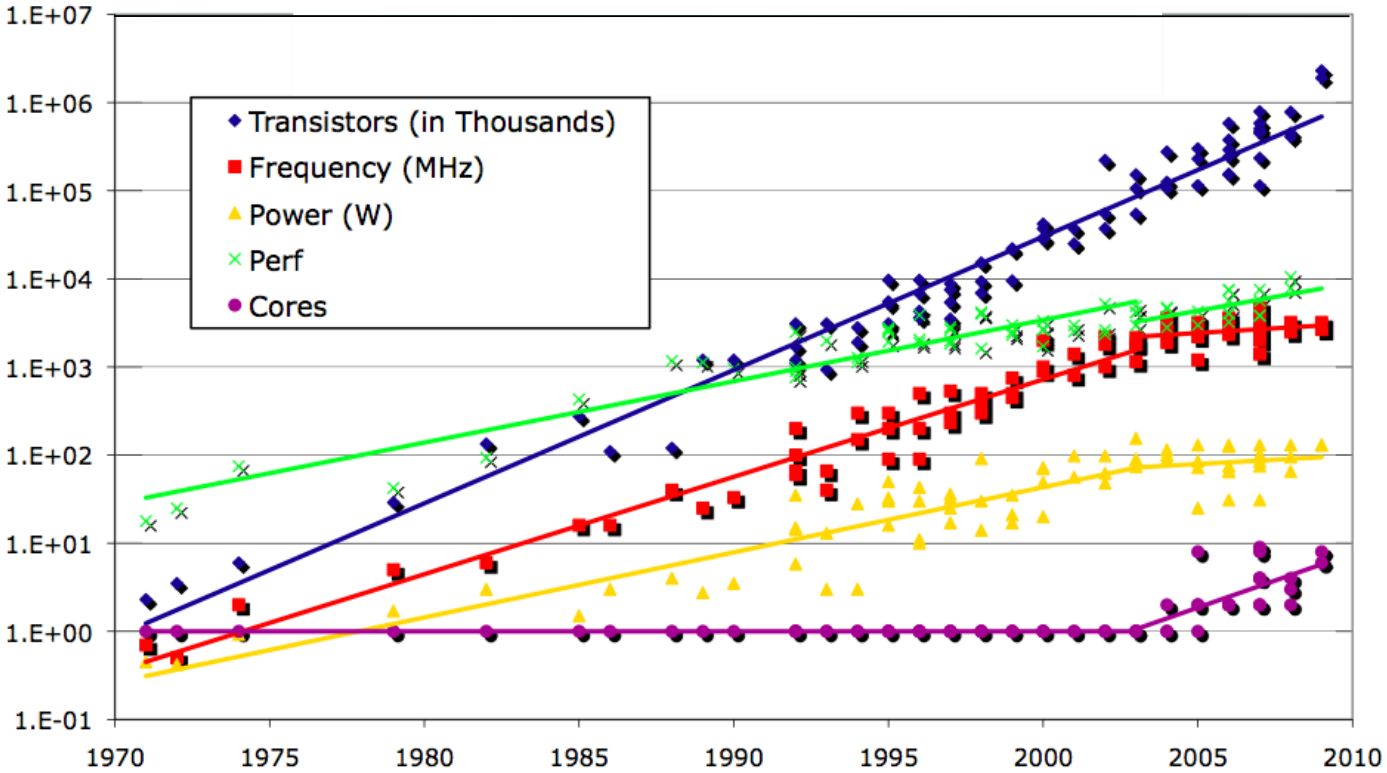
Why parallelism on 2000-present?

- Power consumption is putting a hard technological limit
- Diminishing returns when trying to use transistors to exploit more instruction-level parallelism
- To scale performance, put many processing cores (CPU) on the microprocessor chip instead of increasing clock frequency and architecture complexity
  - Each generation of Moore's law potentially doubles the number of cores
  - This vision creates a desperate need for all computer scientists and practitioners to be aware of parallelism[1]

[1] Parallelism and parallel computing has been taught for several decades in some master and PhD curricula, oriented to solve computationally intensive applications in science and engineering with problems too large to solve on one computer (use 100s or 1000s)

Uniprocessor and multicore performance evolution[2]

The solution to the power consumption: more than one core



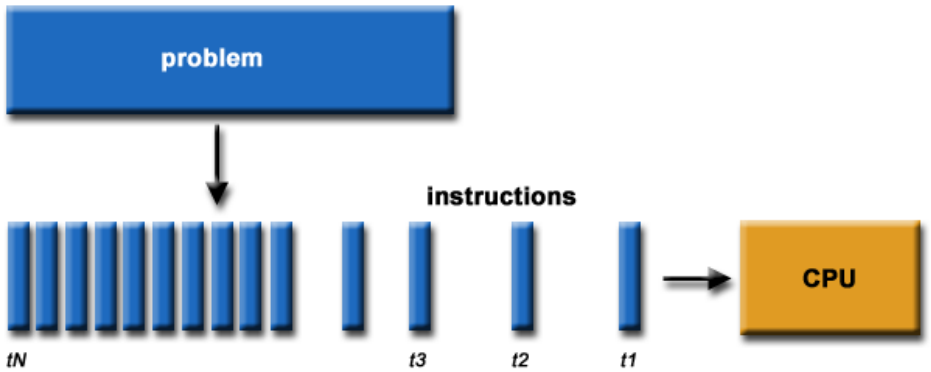
[2] Data collected by M.Horowitz et al.

Outline

Serial execution

Traditionally, programs have been written for serial execution

- To be run on a computer with a single processor (CPU[3])
- Program is composed of a sequence of instructions
- Instructions are executed one after another, only one at any moment in time



[3] Here we mean in-order and not superscalar processor

Concurrent execution

Exploiting concurrency consists in breaking a problem into discrete parts, to be called tasks, to ensure their correct simultaneous execution

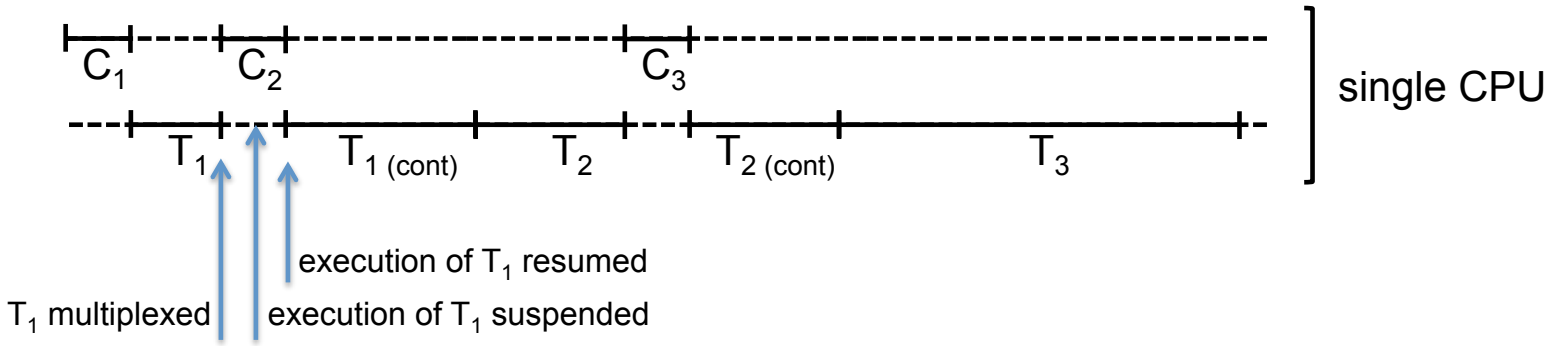
- Concurrent execution assumes that each task is serially executed and that multiple tasks multiplex (interleave) their execution ...
- ... on a single CPU (although it can be on multiple CPU)

Need to manage and coordinate the execution of tasks, ensuring correct access to shared resources

Example: client/server application

Client connection implies the execution of the client task (C). As a response, the server task (T) is executed

Task C_k: receives client requests
Task T_k: executes a single bank transaction (e.g. withdraw some money if identification is correct and balance is sufficient)

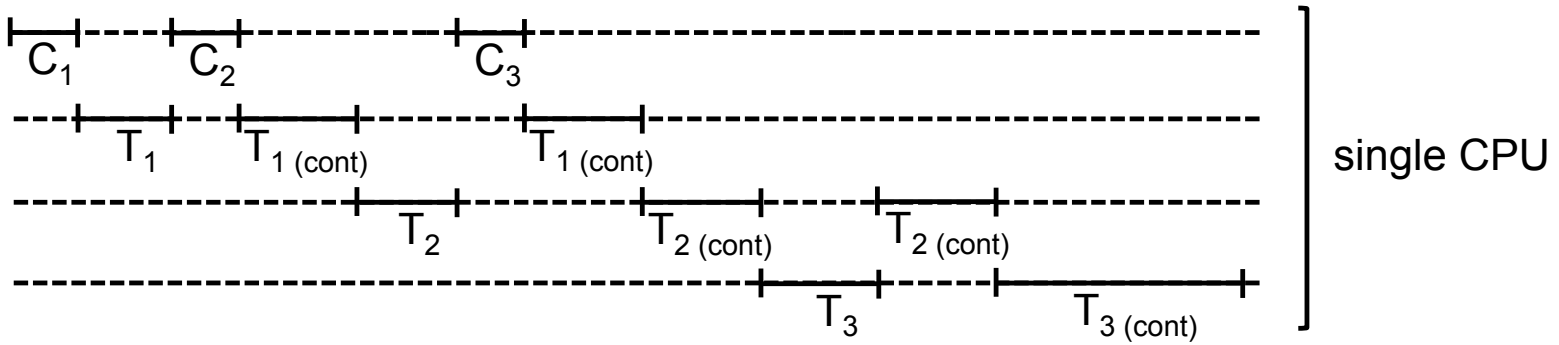


Concurrent execution of client and server tasks

Example: client/server application

Concurrent execution of client and **multiple** server tasks

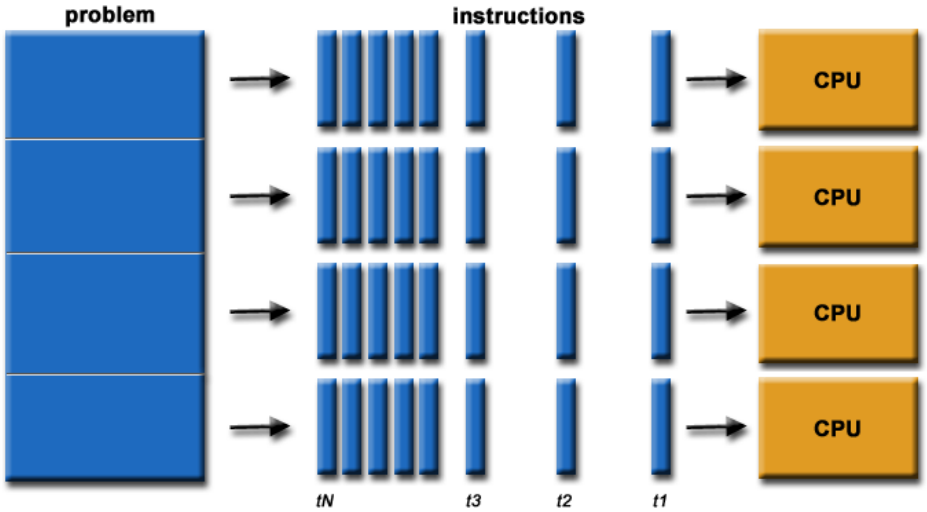
Task C_k: receives client requests
Task T_k: executes a single bank transaction (e.g. withdraw some money if identification is correct and balance is sufficient)



Parallel execution

In the simplest sense, parallelism is when we use multiple processors (CPU) to execute in parallel the tasks identified for concurrent execution

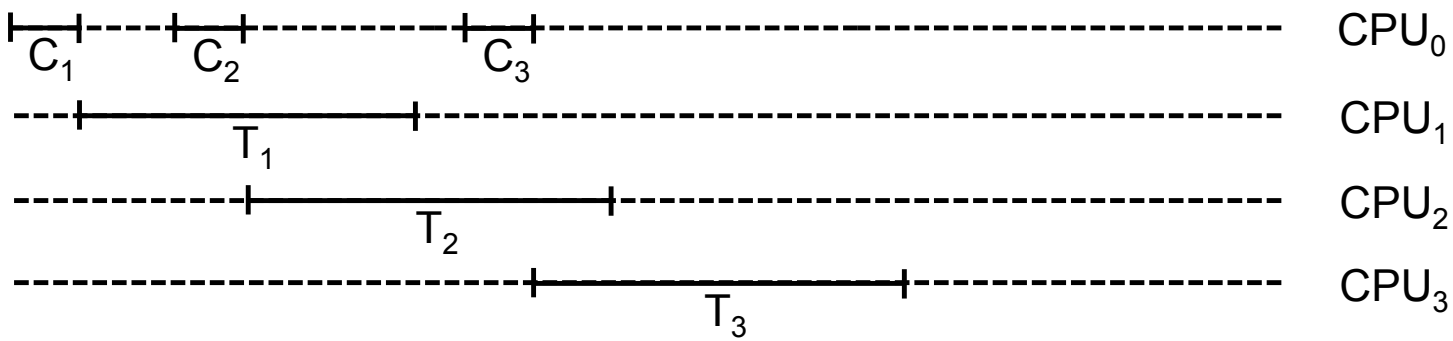
- Ideally, each CPU could receive $\frac{1}{p}$ of the program, reducing its execution time by p



Example: client/server application

Parallel execution of client and multiple server tasks on **several processors**

Task C_k: receives client requests
Task T_k: executes a single bank transaction (e.g. withdraw some money if identification is correct and balance is sufficient)



Throughput vs. parallel computing

Multiple processors can also be used to increase the number of programs executed per time unit

- Throughput computing: Multiple, unrelated, instruction streams (programs) executing at the same time on multiple processors
- n programs on p processors; each program receives $\frac{p}{n}$ processors

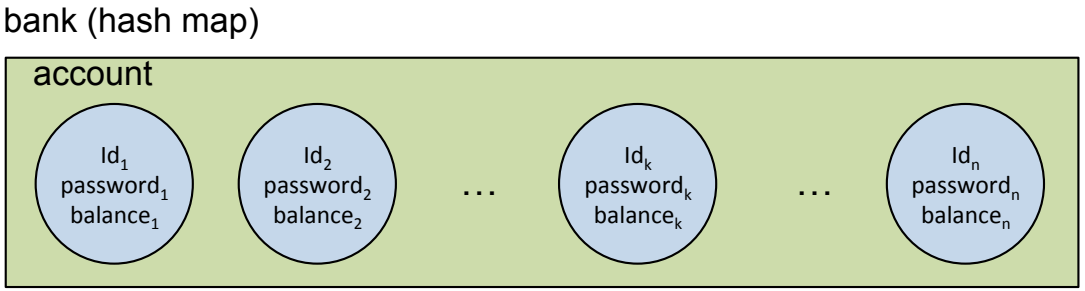
Notice that this is not the same as parallelism, whose objective is to reduce the execution (response) time of a single program:

- Parallel computing: multiple, related, interacting instruction streams (single program) that execute simultaneously
- 1 program on p processors, each processor executes $\frac{1}{p}$ of it

Outline

Examples and potential problems[5]

Bank with several accounts and two terminals/ATMs[4] to operate:



Three methods to operate on an account

- `is_password` to validate if password is correct
- `getbal`: returns the balance in the account
- `post`: deposit/withdraw some money in the account

[4] ATM=Automated Teller Machine

[5] Based on an example used in MIT 6.189 course

Examples and potential problems[6]

We will visit three different examples and show potential problems

- First example: two simultaneous withdraw operations from different/same account
  - Correctness: data race, starvation
- Second example: two simultaneous money transfers from two accounts
  - Correctness: deadlock
- Third example: simple bank statistics
  - Efficiency: lack or dependency of work, overheads, ...

[6] Based on an example used in MIT 6.189 course

Simplified Java code

```
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);
    }
    ...
}
```

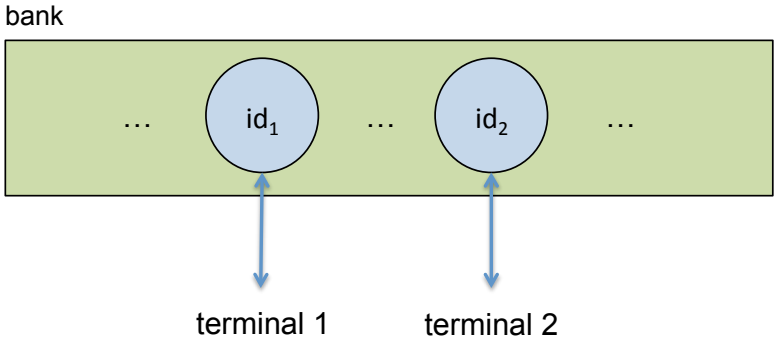
Simplified Java code

```
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();
        }
    }
```

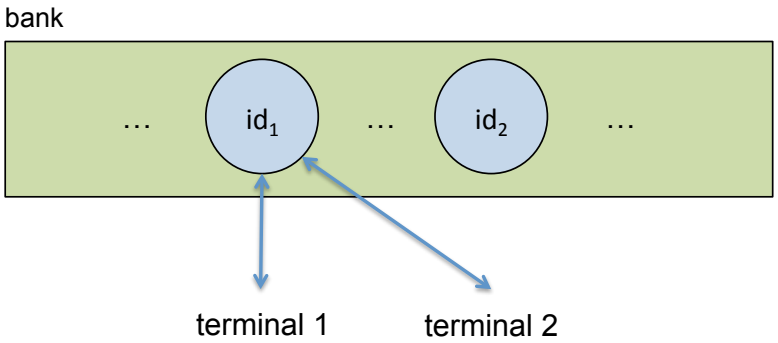
```
    public void run() {
        while(true) {
            try {
                out.print("Account ID > " );
                String id = in.readLine();
                String acc = bnk.get(id);
                if (acc == null) throw new Exception();
                out.print("Password > " );
                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" );
            }
        }
    }
}
```

First example: two simultaneous withdraw operations

- No problem if $id_1 \neq id_2$



- Concurrent execution of `post` method on same account if $id_1 = id_2$: data race



First example: data race and free money

Problem: Data race in the access to `balance`

Time	Client 1 – ATM1	Client 2 – ATM2
1	105 ← GetBal	105 ← GetBal
2	Print 105	Print 105
3	100 ← Read ATM	10 ← Read ATM
4	(GetBal=105)+(-100)>0 → YES	
5		(GetBal=105)+(-10)>0 → YES
	Post(-100) – balance=balance+val	Post(-10) – balance=balance+val
6	Step 1: Read Balance → 105	
7		Step 1: Read Balance → 105
8	Step 2: Sum → 105 + (-100)	
9	Step 3: Write Balance → 5	
10		Step 2: Sum → 105 + (-10)
11		Step 3: Write Balance → 95

First example: two withdraw operations, same account

Synchronized methods in Java execute the body as an atomic unit

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

First example: data race and free money, red numbers!

Problem: Data race still exists in the access to `balance`

Time	Client 1 – ATM1	Client 2 – ATM2
1	105 ← GetBal	105 ← GetBal
2	Print 105	Print 105
3	100 ← Read ATM	10 ← Read ATM
4	(GetBal=105)+(-100)>0 → YES	
5	Lock method	
	Post(-100) – balance=balance+val	
6	Step 1: Read Balance → 105	
7		(GetBal=105)+(-10)>0 → YES
8	Step 2: Sum → 105 + (-100)	Locked in method
9	Step 3: Write Balance → 5	
10		
11	Unlock method	Unlocked from method
12		
		Post(-10) – balance=balance+val
13		Step 1: Read Balance → 5
14		Step 2: Sum → 5 + (-10)
15		Step 3: Write Balance → -5

First example: two withdraw operations, same account

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

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

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

First example: two withdraw operations, same account

```
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" );
          }
      }
   }
}
```

Problem: balance may be initially OK, but could not withdraw!

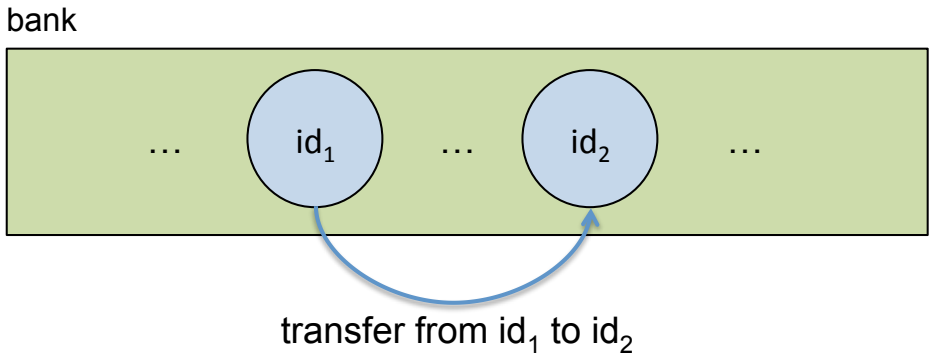
First example: two withdraw operations, same account

```
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" );
          }
      }
  }
}
```

Problem: Starvation if one ATM does not release acc

Second example: bank transfers

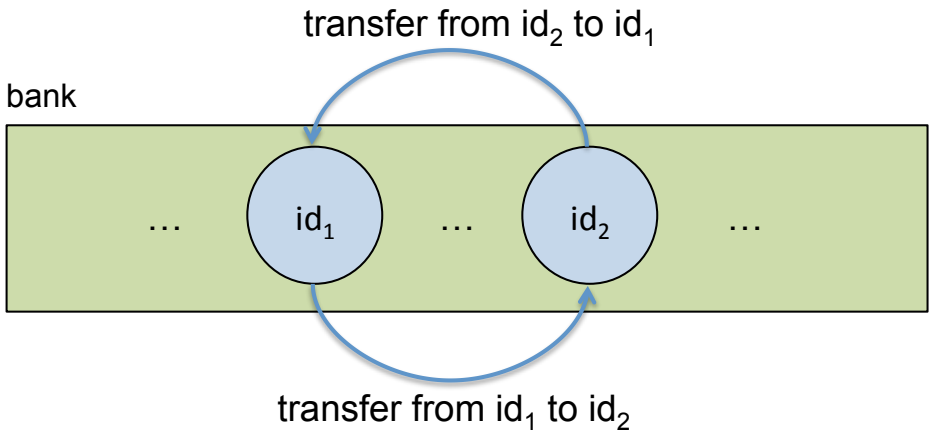


We need to protect the execution of `post` methods

```
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);
      }
   }
}
```

Second example: two simultaneous transfers, 2 accounts

But, what if "John wants to transfer $10 to Peter's account" while "Peter wants to also transfer $20 to John's account"?



Both get blocked in the `synchronized(to)` construct

- Cycle in locking graph = deadlock

Second example: deadlock

Time	John – ATM1	Peter – ATM2
1	Lock (John account)	
2		
3		Lock (Peter account)
4		Locked on (John account)
5	Locked on (Peter account)	
6	DEADLOCK	
7		
8		
9		
10		
...		

Second example: two simultaneous transfers, 2 accounts

Standard solution: canonical order for locks (i.e. acquire in increasing order, release in decreasing order)

```
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);
            }
        }
}
```

Summary: potential concurrency problems

Race Condition

- Multiple tasks read and write some data and the final result depends on the relative timing of their execution

Starvation

- A task is unable to gain access to a shared resource and is unable to make progress

Deadlock

- Two or more tasks are unable to proceed because each one is waiting for one of the others to do something

Livelock (see philosophers problem)

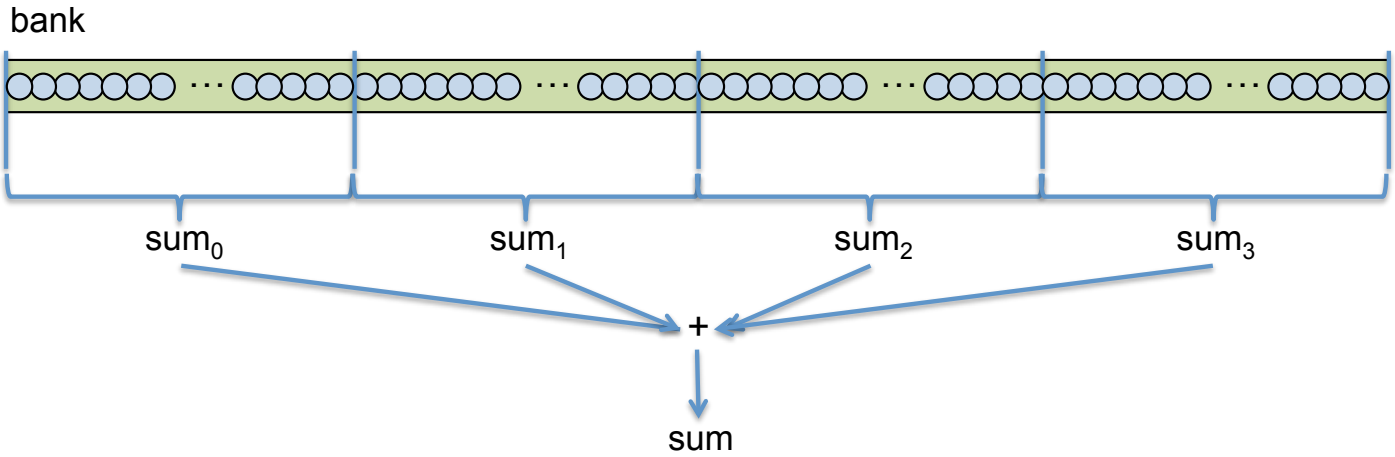
- Two or more tasks continuously change their state in response to changes in the other tasks without doing any useful work

Third example: bank statistics

- Imagine that every day the bank needs to compute the total interest that has to pay to all its customers (hundred thousands, or even millions!)
- We need to perform the Dot Product of two vectors
  - $sum = \sum_{i=1}^{number\_clients} balance_i \times interest_i$

Third example: bank statistics

- The computation and data can be partitioned among multiple processors (P), each working with $1/P$ elements and accumulating the result in a "shared" variable (e.g. `sum`)



  - Computation time approx. divided by P

Third example: bank statistics

```
class Task extends Thread {
   Task(double[] a, double[] b,
        int iStart, int iEnd) {
      this.a = a; this.b = b;
      this.iStart = iStart;
      this.iEnd = iEnd;
      start();
   }

   double[] a, b;
   int iStart, iEnd;
   double sum;

   public void run() {
      sum = 0;
      for (int i = iStart; i < iEnd; i++)
         sum += a[i] * b[i];
   }

   public double getSum() {
      try {
         join();
      } catch(InterruptedException e) {}
      return sum;
   }
}
```

```
import java.util.*;

class DotThreads {
  public static void main(String[] args) {
    int n = 1024*1024*1024;
    double[] balance = new double[n];
    double[] interest = new double[n];

    /* Initialize/copy vectors */

    int numThreads = 4;
    List<Task> tasks = new ArrayList<Task>();
    for (int i = 0; i < numThreads; i++)
       tasks.add( /* Creation */
         new Task(balance, interest,
                  i * n / numThreads,
                  (i + 1) * n / numThreads));

    double sum = 0;
    for (Task t : tasks)
       sum += t.getSum(); /* wait */

    System.out.println("a*b = " + sum);
  }
}
```

Potential parallelism problems

- Lack or dependency of work
  - Coverage or extent of parallelism in algorithm
  - Dependencies (sequential is an extreme case)
  - Hard to equipartition the work
    - Load imbalance
  - Due to the parallelization strategy and parallel programming model
- Overheads of the parallelization
  - Granularity of partitioning among processors
    - Work generation and synchronization
  - Locality of computation and communication

Parallelism (PAR)

Introduction and motivation

Eduard Ayguadé, Josep Ramon Herrero and Daniel Jiménez

Computer Architecture Department
Universitat Politècnica de Catalunya

2014/15-Fall