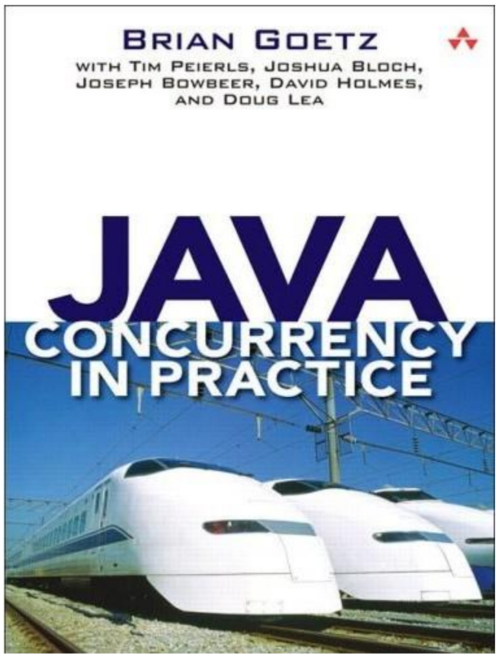
Concurrent Programming

Recommended Textbook



Download and investigate source code examples
http://jcip.net/

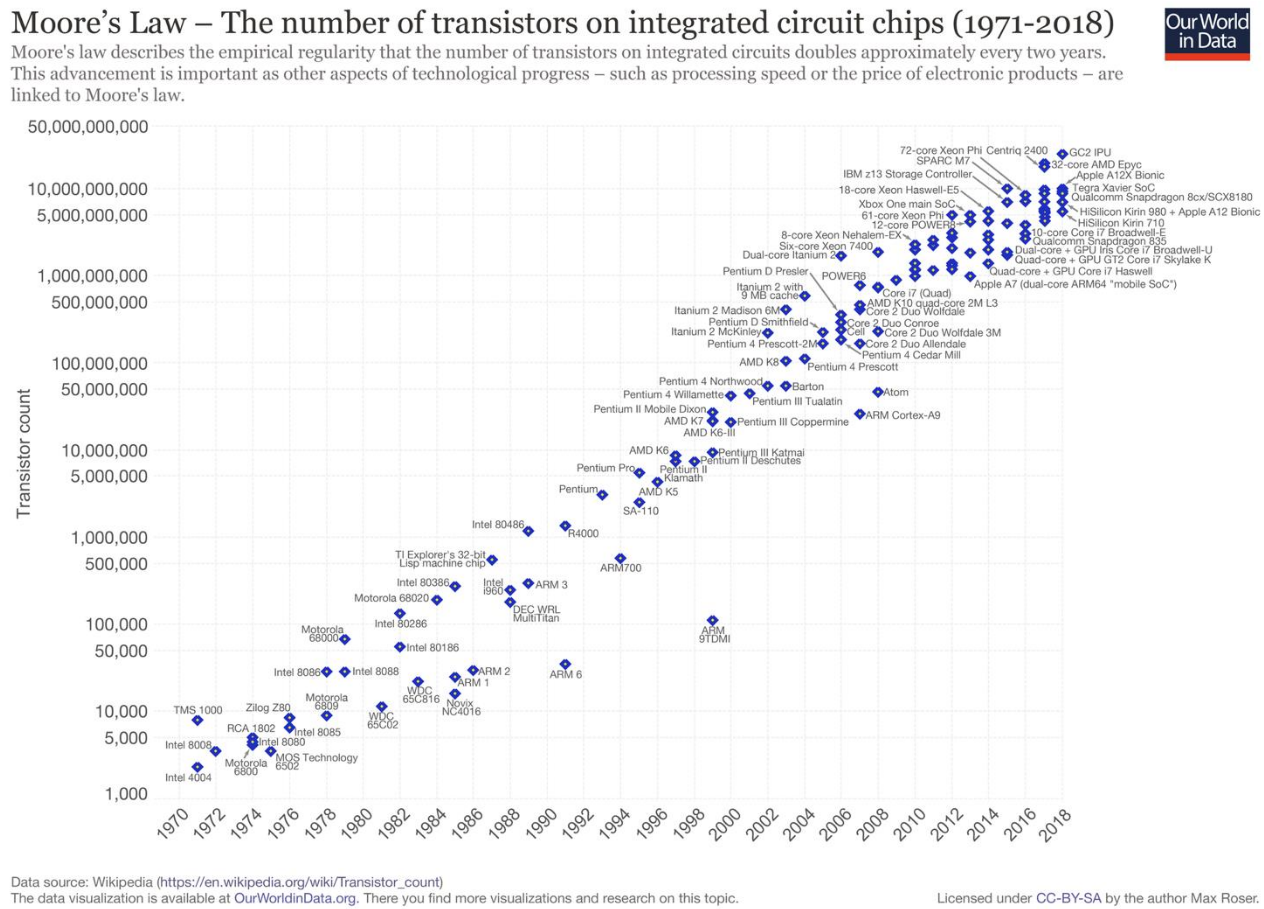
Concurrency?

- = “multi-threading”
  - single-threaded: at any point during execution, at most one instruction can be executed next.
  - In multi-threaded applications, several instructions can be executed “next”!
- Programming languages include mechanisms for concurrency
  - Threads
  - Locks
  - Interrupts
  - Etc.

Why Concurrency?

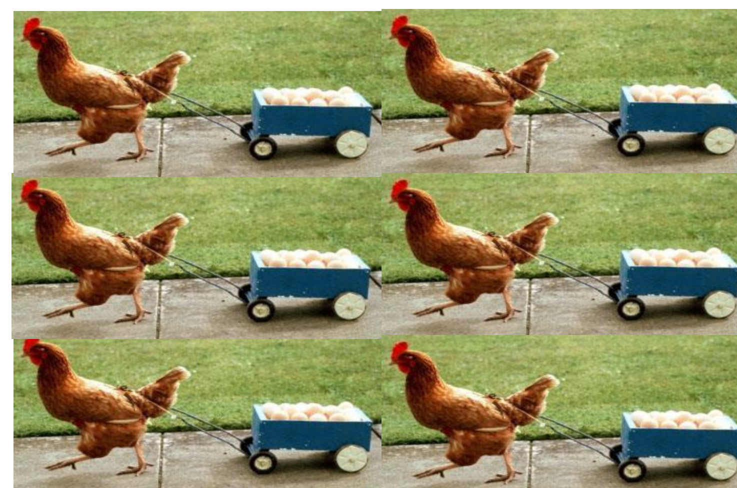
- Performance
  - If they can do operations simultaneously, applications run faster!
- Availability
  - Compute-intensive parts of application need not slow down other parts (e.g. user interface)
- Application demands
  - Many applications feature concurrency as part of system design (e.g. operating systems, communications protocols, simulations)

Why Concurrency? (cont.)



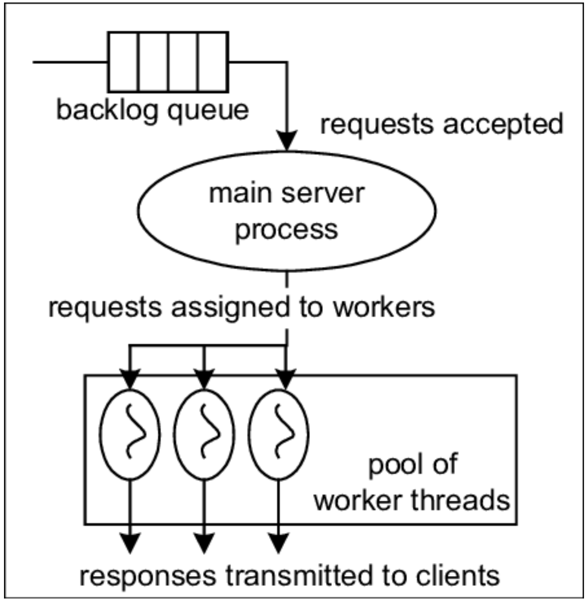
Why Concurrency? (cont.)

- Exploiting multiple processors
  - Processor speeds are not increasing as fast as they used to
- Multi-CPU machines becoming standard
- Can't take full advantage of multiple CPUs without concurrent software



Why Concurrency? (cont.)

- For some problems, concurrency provides a very natural programming model
  - For example, problems involving many, largely independent actors or actions, e.g.,
    - Simulations:
      - run multiple simulations with different parameters
    - Compute servers:
      - web servers, email servers



Why Concurrency? (cont.)

- Isolates and simplifies tasks
- For instance servers typically interact with multiple clients
  - High performance, non-concurrent implementations have to multiplex (switch between) clients
  - Concurrent servers can handle each client in a separate thread of control

Threads are everywhere

- Even if your program never explicitly creates a thread, framework may create threads on your behalf, and code called from these threads must be safe. So thread is NOT optional.
  - Garbage Collector
  - Finalizer
  - AWT, Swing
    - Deferred tasks
  - Servlet, RMI
    - Creates pools of threads and invoke them

Why Not Teach It Sooner?

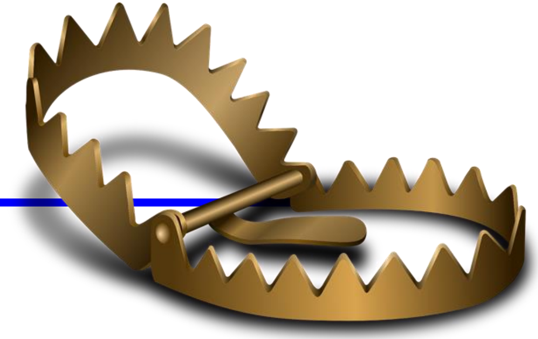
- We do!
- However, concurrency is hard
  - Concurrent programs are hard to debug
    - breakpoint?
  - Concurrent programs are hard to optimize

```
a = 1;
while(true){
    if(a < 1) break;
    …
}
```

Not infinite loop. Other threads may update a

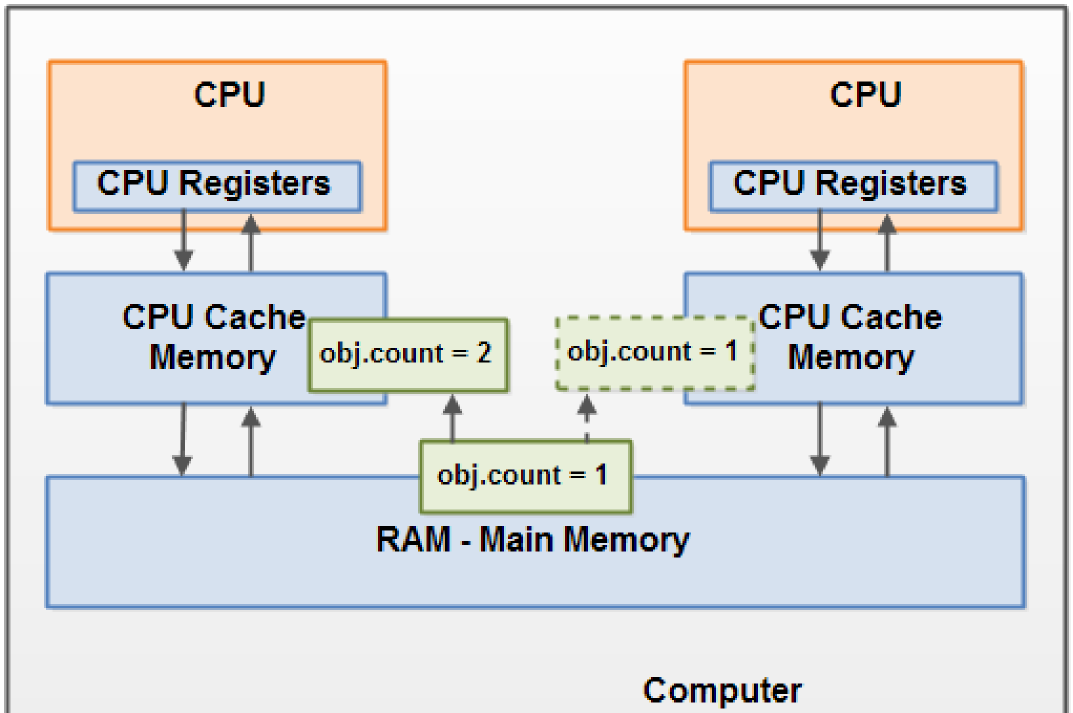
  - Concurrent programs are hard to test
    - interleave

Why Is Concurrency Hard?



- Nondeterminism!
  - Executing same program can yield different answers
  - Replaying a given execution is very difficult
- Concurrency breaks *procedural abstraction*
  - Procedural abstraction: a given sequence of instructions will always return the same result if started in the same state
  - Implication: you can think of a sequence of instructions as a single “big instruction”
  - Basis for: compilation, method definition, etc.

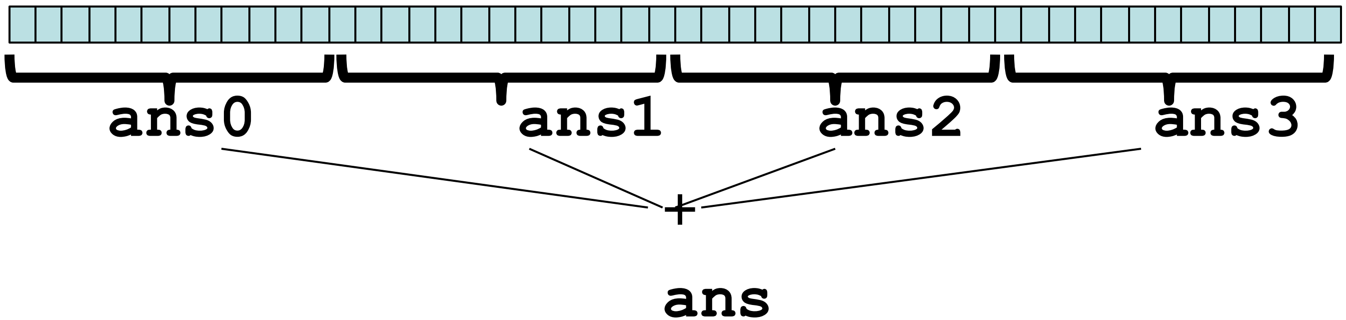
Visibility


CPU
CPU Registers
CPU
CPU Registers
CPU Cache
Memory
obj.count = 2
obj.count = 1
CPU Cache
Memory
obj.count = 1
RAM - Main Memory
Computer

Concurrency is cool

Parallel Sum Example

- Sum elements of a large array
- Idea: Have 4 simultaneous tasks each sum 1/4 the array



- Create 4 *threads*, assigned a portion of the work
- *Wait* for each object to finish using **join()**
- Sum 4 answers for the *final result*

Parallel Sum Example

```
class SumThread extends Thread {
  int lo; // arguments
  int hi;
  int[] arr;

  int ans = 0; // result

  SumThread(int[] a, int l, int h) {
    lo=l; hi=h; arr=a;
  }

  public void run(){
    for(int i=lo; i < hi; i++)
      ans += arr[i];
  }
}
```

Parallel Sum Example

```
int sum(int[] arr){
  SumThread[] ts = new SumThread[4];

  int len = arr.length; // do parallel computations
  for(int i=0; i < 4; i++){
    ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
    ts[i].start ();
  }

  int ans = 0; // combine results
  for(int i=0; i < 4; i++)
    ans += ts[i].ans;
  return ans;
}
```

`ts[i]` is still running

Parallel Sum Example

```
int sum(int[] arr){
  SumThread[] ts = new SumThread[4];

  int len = arr.length; // do parallel computations
  for(int i=0; i < 4; i++){
    ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
    ts[i].start ();
  }
  int ans = 0; // combine results
  for(int i=0; i < 4; i++){
    ts[i].join();  //wait for threads to finish
    ans += ts[i].ans;
  }
  return ans;
}
```

Speed Up

Sequentil sum:9.69964730357349E7
parallel sum:9.69964730357352E7
Sequentil sum time:1874
parallel sum time:352

Speed up: 5.32

Running a Sequential Program

- Executable

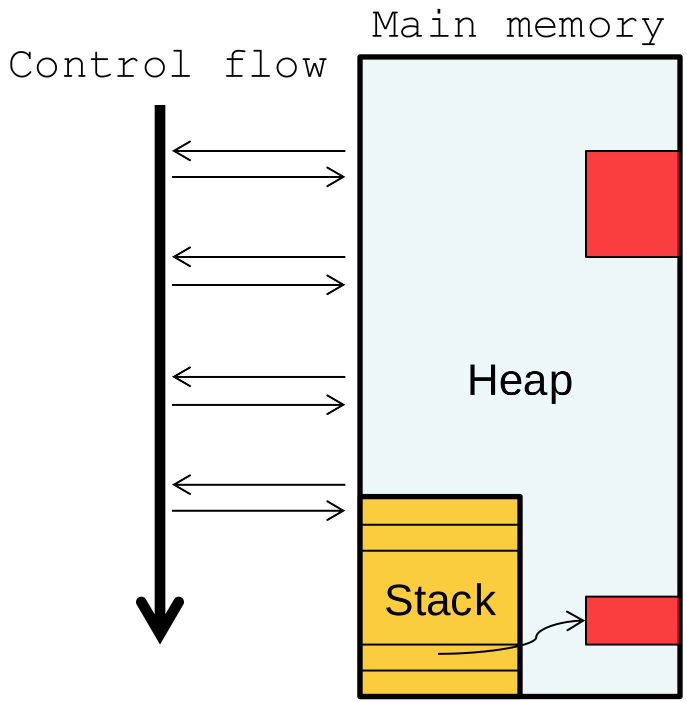
  Machine instructions to be performed
- Program counter

  Next instruction to be executed
- Stack

  Current variable definitions
- Heap

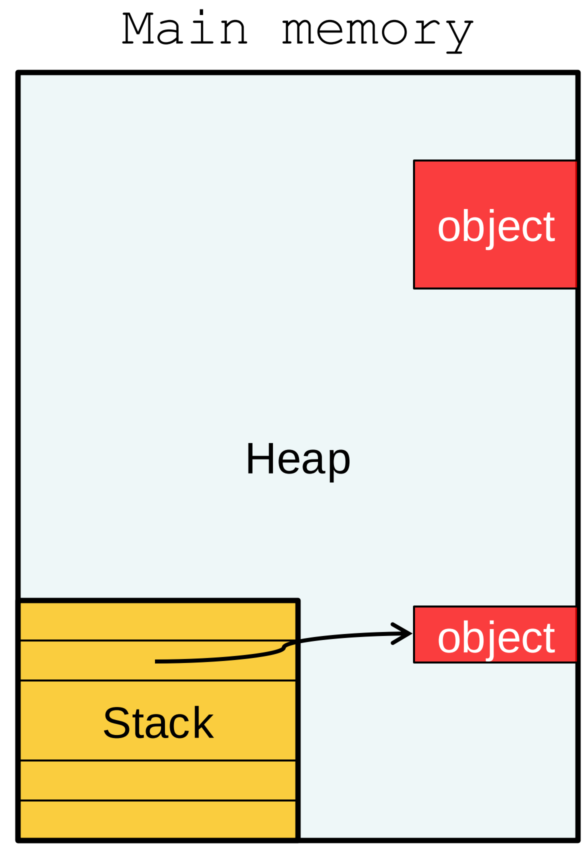
  Dynamically allocated data structures
- Control flow

  Sequence of instructions performed during an execution



Java Memory Model

- Stack
  - Local variables
  - Method parameters
- Heap
  - Objects!
  - Every call to **`new`** allocates space on heap
- Class-typed variables
  - contain either null or a reference to heap



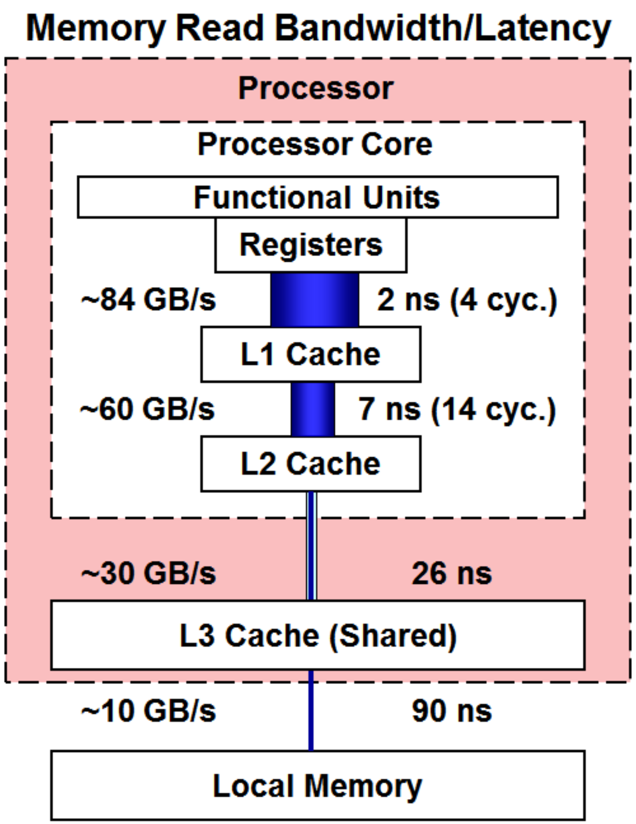
More on Main Memory (MM)

- Naively, MM is a table:
  - Each address can store a value
  - Each address refers to one memory location (no copies)

Address	Value
0000	‘a’
0001	37
0002	NULL

- In reality, several copies of a given address are possible
  - Caches
  - Registers
  - …
- Why? *Performance*
  - Higher-speed memory is more expensive
  - Copying frequently used data into high-speed memory (register, cache) improves performance while containing cost

Memory Latency



Concurrent Programs

- Multiple control flows!
- Programs with multiple control flows can be
  - Concurrent
  - Parallel
  - Distributed
- Control flows are either
  - Processes
  - Threads

Concurrent vs. Parallel vs. Distributed

- Concurrent

  number of control flows unrelated to number of physical processors

- Parallel

  number of control flows ≤ number of physical processors; each flow has its own processor

- Distributed

  Multiple machines connected via network

An analogy

A program is like a recipe for a cook

- One cook who does one thing at a time! (*Sequential*)

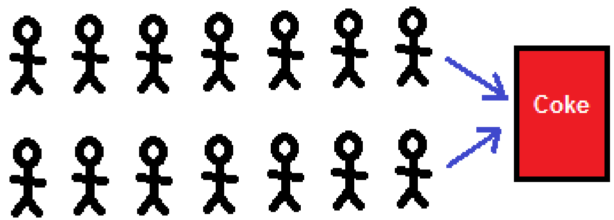
Parallelism:

- Have lots of potatoes to slice?
- Hire helpers, hand out potatoes and knives
- But too many chefs and you spend all your time coordinating

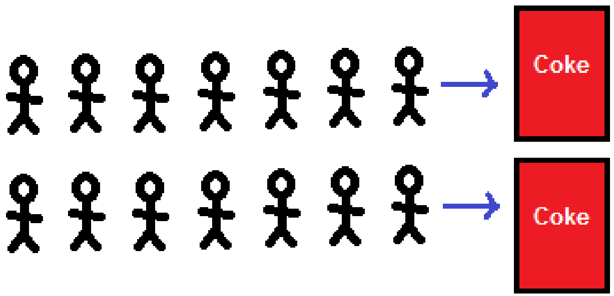
Concurrency:

- Lots of cooks making different things, but only 4 stove burners
- Want to allow access to the burners, but not cause spills or incorrect burner settings

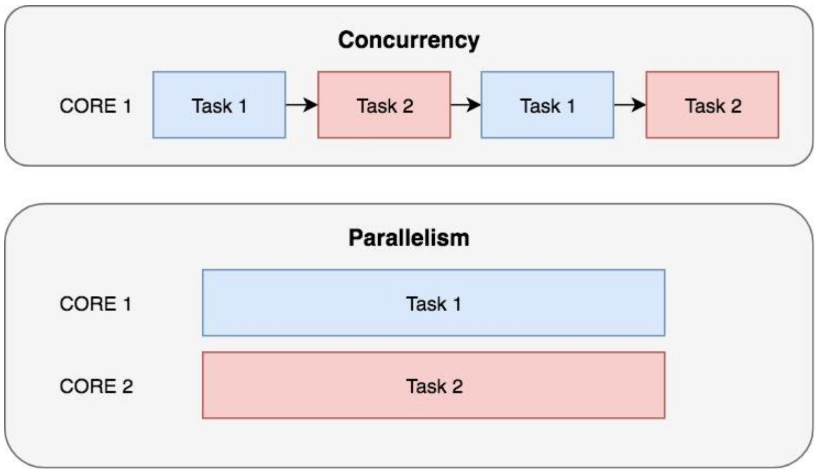
An analogy



Concurrent: 2 queues, 1 vending machine



Parallel: 2 queues, 2 vending machines



Parallelism Example

Parallelism: Use extra computational resources to solve a problem faster

Pseudocode for array sum

```
int sum(int[] arr){
  res = new int[4];
  len = arr.length;
  for(i=0; i < 4; i++) { //parallel iterations
    res[i] = sumRange(arr, i*len/4, (i+1)*len/4);
  }
  return res[0] + res[1] + res[2] + res[3];
}

int sumRange(int[] arr, int lo, int hi) {
   result = 0;
   for(j=lo; j < hi; j++)
      result += arr[j];
   return result;
}
```

Concurrency Example

Concurrency: Correctly and efficiently manage access to shared resources

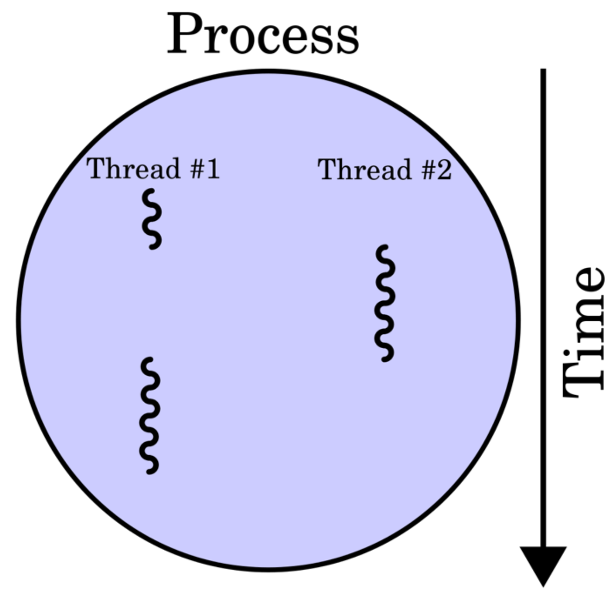
Pseudocode for a shared chaining hashtable

- Prevent *bad interleavings* but allow some concurrent access

```
class Hashtable<K,V> {
  …
  void insert(K key, V value) {
    int bucket = …;
    prevent-other-inserts/lookups in table[bucket];
    do the insertion
    re-enable access to table[bucket];
  }
  V lookup(K key) {
     allow concurrent lookups to same bucket
  }
}
```

Processes vs. Threads

- Process
  - A process is a unit of resource allocation & protection
- Thread
  - A Java thread is a unit of computation that runs in the context of a process

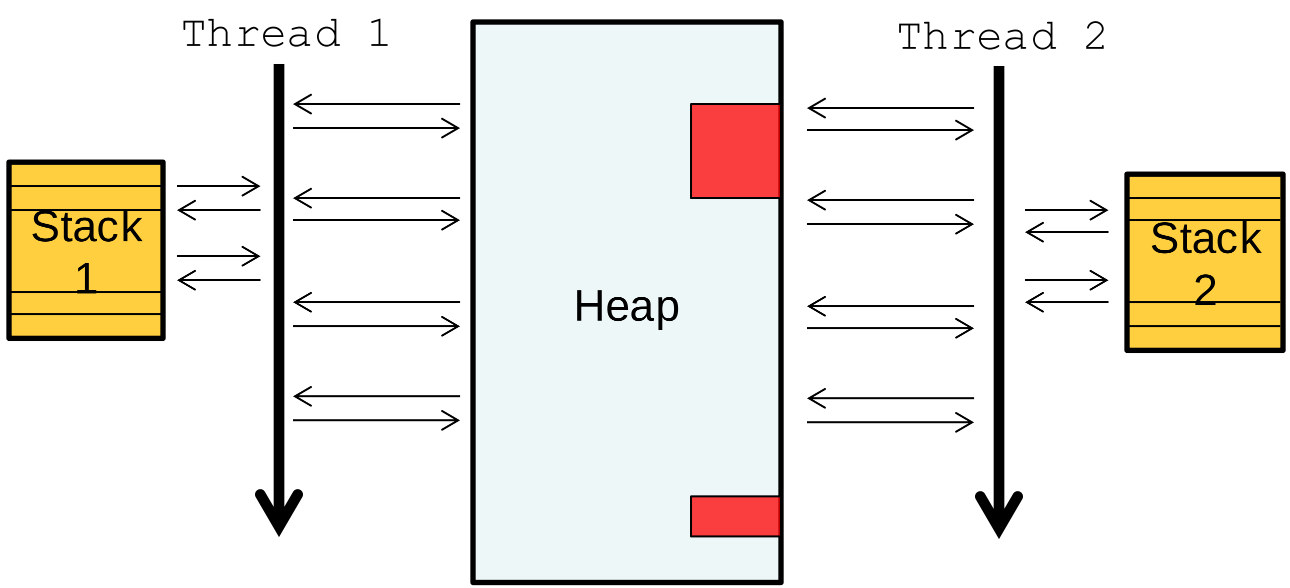


Processes vs. Threads

- Processes
  - Possess own heap
  - Communicate via *IPC* (= inter-process communication) mechanisms
    - Sockets
    - Message passing
    - Etc.
- Threads
  - Contained within processes
  - Possess own stack, program counter
  - Share heap with other threads in same process
  - Communicate via shared memory
- Historically
  - Process management handled by operating system
  - Processes were single-threaded

Multi-threaded Process

Java threads running in the same process can communicate with each other via shared objects or message passing



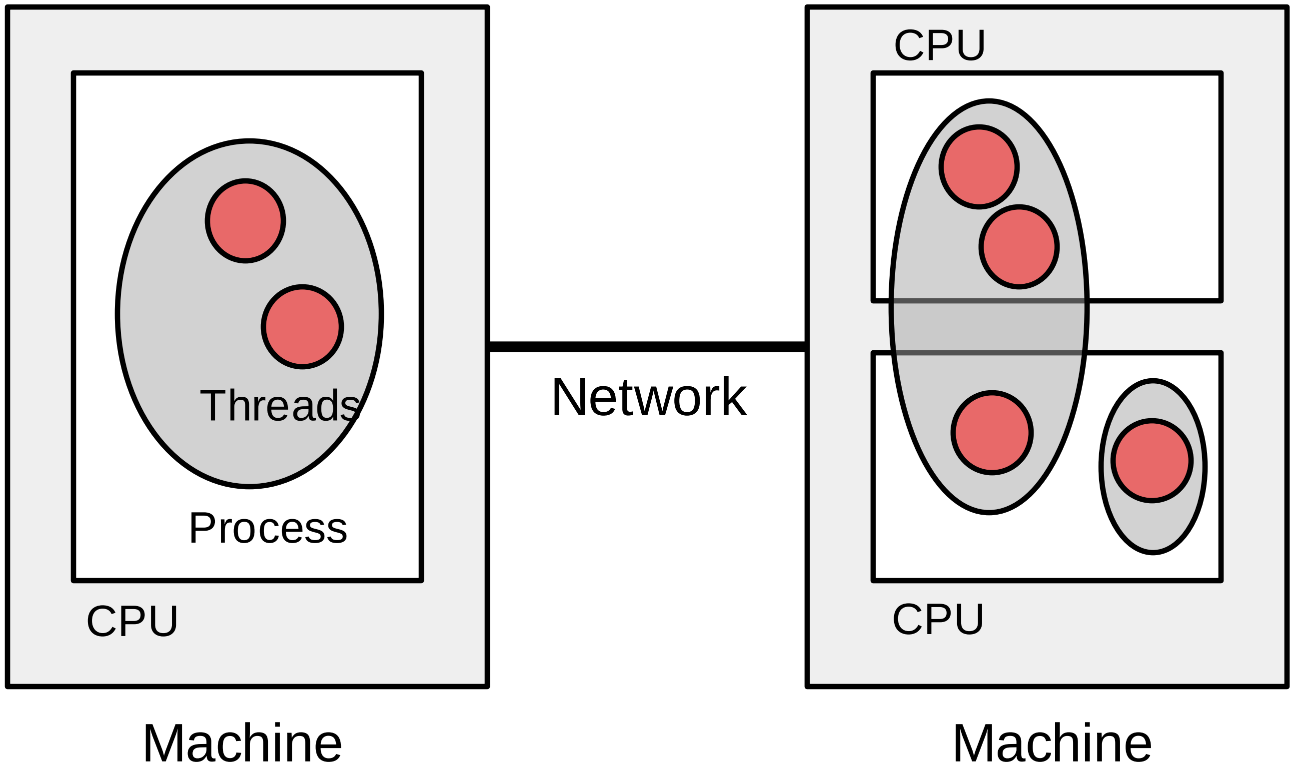
Running a Multi-Process/Multi-Threaded Application

- Running a thread requires using a processor
- What decides which thread gets which processor?
  - Scheduler (part of operating system)!
  - Scheduling policy decides which threads run when
  - Pre-emptive schedulers can interrupt one thread and let another run on a given processor
    - Interrupted thread is "suspended": its stack, program counter are saved so that thread can be re-activated later
    - Stack, program of new thread are loaded and new thread activated
    - This is called a *context switch*

Threads, Processes and Processors

- Do processes run on single machine? Yes
- Do processes run on a single processor? Not necessarily
  - Different threads can run on different processors
  - Scheduler makes this decision
- Do threads run on a single processor?
  - Usually
  - Some schedulers support *thread migration* (why?)

A Reference Model for Distributed / Parallel / Concurrent Programs



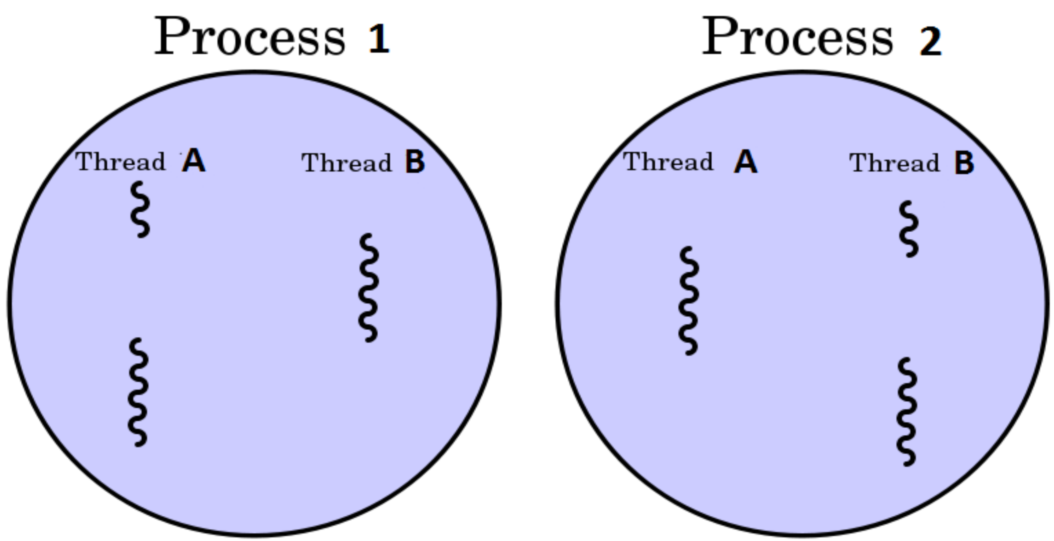
Language Support for Concurrency

- Many languages support concurrency!

  C, C++, C#, OCaml, Java, Scala, Erlang, Python, …
- Traditionally: process / thread management handled via system calls to operating system
  - Not part of core language (e.g. C)
  - Platform-specific, non-portable, since different OS's have different mechanisms
- Modern languages (e.g. Java, Scala, Erlang) include mechanisms for thread management directly

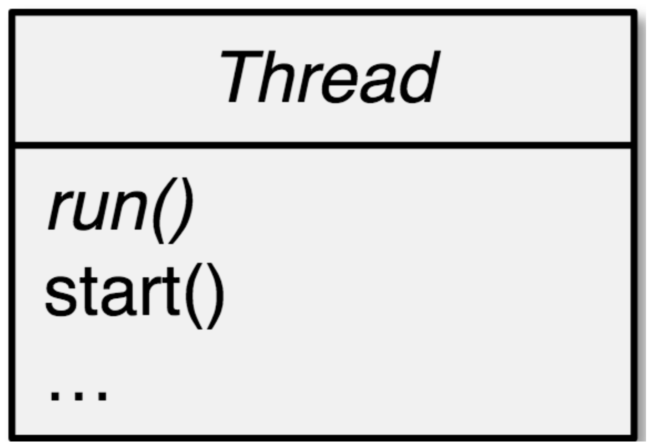
Java Threads

Threads are the most basic way of obtaining concurrency in Java



Java Threads Are Objects

- Object class is **`Thread`**, which is part of java.lang package (automatically imported!)



Java Threads

- Thread objects include:
  - `public void run()` executed when thread is launched
  - `public void start()` to launch the thread
  - Other methods that we will study later

  - Constructors, of which more later, but here are two:
    - `Thread()` create a thread
    - `Thread(String name)` create a thread with the given name

Giving Code to Java Threads

```
public class Worker extends Thread {
   public void run() {


      // code to run goes here

   }
}
```

Override the run() method in the subclass &
define the thread's computations

Giving Code to Java Threads

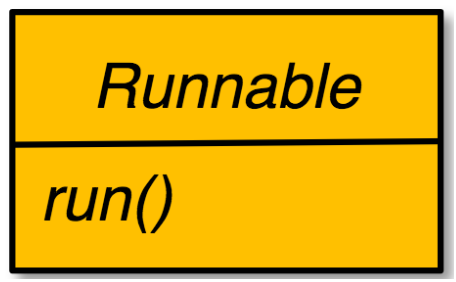
```
public class Worker extends Thread {
   public void run() {
     // code to run goes here
   }
}
```

Create & start a thread using a named subclass of Thread

```
Thread t =  new Worker();
t.start();
```

Implement the Runnable interface

- Another approach:
  - Implementing `Runnable` interface
  - override **`run()`**



“Desired Functionality in `run()`”?

- Define a class implementing the **`Runnable`** interface
- Implement the run() method of an interface to define the thread’s computations

  - Use relevant constructor in **`Thread`** on objects in this class

```
Thread (Runnable target)
Thread (Runnable target, String name)
```

Implementing Runnable Interface

```
public class Worker implements Runnable{
  public void run () {
     // code to run goes here
  }
}
```

Create an instance of a named class as the runnable

```
Runnable r = new Worker();
```

Implementing Runnable Interface

```
public class Worker implements Runnable{
  public void run () {
     // code to run goes here
  }
}
Runnable r = new Worker();
```

Pass that runnable to a new thread object & start it

```
new Thread(r).start();
```

Anonymous Inner Class

Create & start a thread using an anonymous inner class as the runnable

```
new Thread(new Runnable() {
    public void run(){
        // code to run goes here
    }
}).start()
```

Java 8: Lambda Expression

```
new Thread(() -> {
    // code to run goes here
}).start();
```

Java 8: Lambda Expression

You can name the Runnable:

```
Runnable r = () -> {
    // code to run goes here
};
new Thread(r).start(
```

Thread Implementation via Subclassing (Inheritance)

```
public class HelloWorldThread extends Thread {
  public void run() {
    System.out.println ("Thread says Hello World!");
  }
}
```

New class **`HelloWorldThread`** is introduced

- Extends **`Thread`** class
- Uses overriding to redefine **`run()`** method to do what we want

Thread Implementation via Runnable

```
public class HelloWorldRunnable implements Runnable{
  public void run() {
    System.out.println ("Runnable says Hello
                                World!");
  }
}
```

- **`Runnable`** is an interface in java.lang containing only: **`public void run()`**
- This class implements `Runnable` by providing each object with a **`run()`** method
- Constructor for `Thread` class can now be called with objects in this class

Thread Creation

```
Thread h1 = new HelloWorldThread ();
Thread h2 = new Thread (new HelloWorldRunnable ());
h1.start();
h2.start();
```

- `h1` is thread object created from subclass of **`Thread`**
- `h2` is thread object created from **`Runnable`** object
- Output is two instances of “Hello World!”

Passing Parameters to a Java Thread

Pass parameters as parameters to a class constructor

```
public class Worker extends Random
                                implements Runnable {
    private int count;
    public Worker(int c) {
        count = c;
    }

    public void run () {
     for(int i = 0; i < count; i++){
         System.out.println("Thread id " +
                                 Thread.currentThread();
     }
    }
}
```

Passing Parameters to a Java Thread

Pass parameters as parameters to a class constructor

```
new Thread(new Worker(10))
```

Passing Parameters to a Java Thread

Pass parameters as parameters to "setter" methods

```
public class WorkerThread extends Thread{
    private int count;
    public WorkerThread setCount(int c) {
        count = c;
        return this;
    }

    public void run () {
     for(int i = 0; i < count; i++){
        System.out.println("Thread id " +
                              Thread.currentThread();
     }
    }
}
```

Passing Parameters to a Java Thread

Use the fluent interface to pass parameter(s) when the thread is created

```
Thread thread = new MyThread().setCount(10);
```

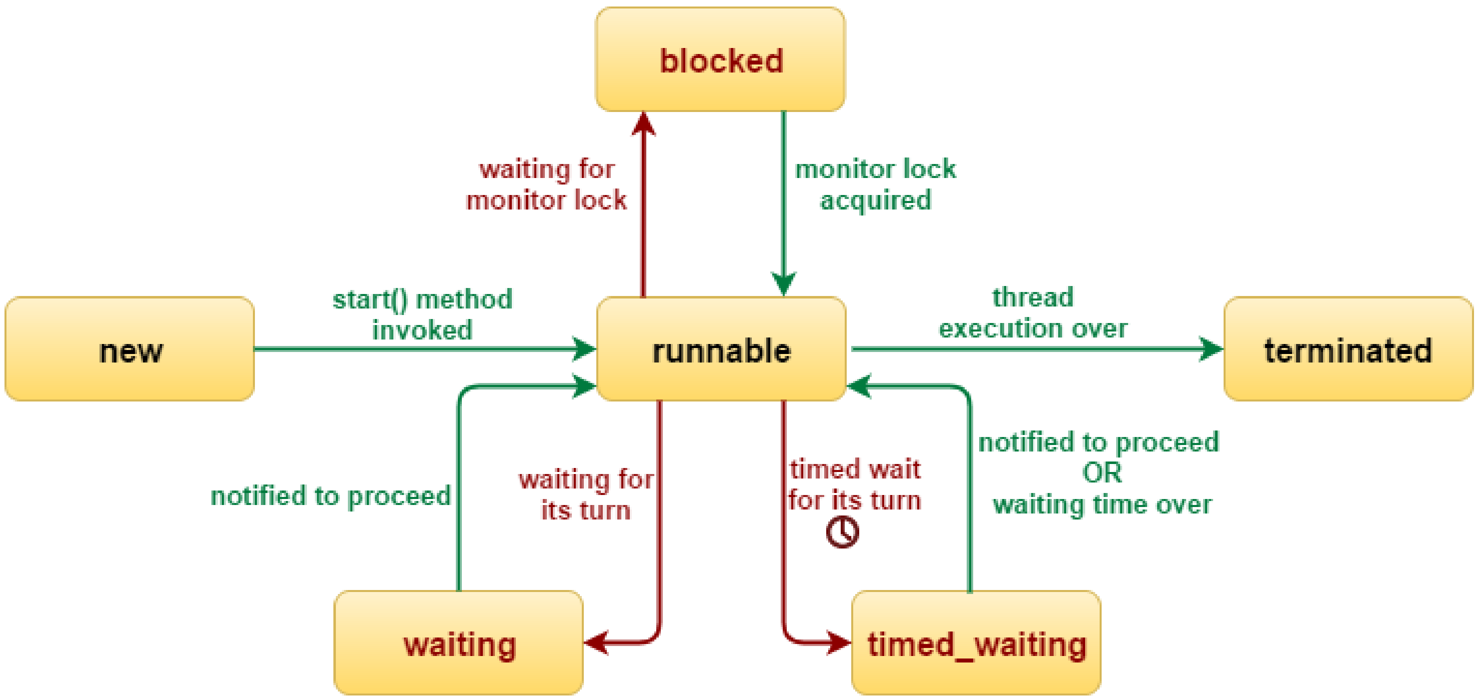
Returns MyThread

Thread States?

- Accessible via method `Thread.State getState()`
- `Thread.State` is an enumerated type recording state of thread object
  - `NEW`
    A thread that has not yet started is in this state.
  - `RUNNABLE`
    A thread executing in the Java virtual machine is in this state.
  - `BLOCKED`
    A thread that is blocked waiting for a monitor lock is in this state.
  - `WAITING`
    A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
  - `TIMED_WAITING`
    A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
  - `TERMINATED`
    A thread that has exited is in this state.

[Quoted from http://docs.oracle.com/javase/6/docs/api/java/lang/Thread.State.html]

Thread States



More on Thread States

- Some **`Thread`** methods (e.g. **`start()`**) only applicable when object is in correct state
- The states **`NEW`**, **`RUNNABLE`**, **`TERMINATED`** are probably easiest to understand
- We will learn about the states **`BLOCKED`**, **`WAITING`**, **`TIMED_WAITING`** later

Other Thread State Methods

- **`boolean isAlive()`**
  - Returns **`true`** if thread has been started but is not terminated
  - **`t.isAlive()`** equivalent to

    `(t.getState() != NEW) && (t.getState() != TERMINATED)`
- **`void join()`**
  - Blocks until thread terminates, then terminates
  - **`t.join()`** very similar to

    `while (t.getState() != TERMINATED) { }`
- **`void join(int millis)`**

  Like `t.join()` except that if `t` has not terminated in millis milliseconds, then **`t.join(millis)`** nevertheless terminates

Threads and Process Termination

- A process (JVM) terminates when “there is nothing left that has to be done”
- When does this hold?
  - When the main thread terminates?
  - When all threads terminate?
  - When “the important” threads terminate?
- A Java process can terminate if and only if all user threads (including, but not only, main) have terminated

User Threads vs. Daemon Threads

- Threads may be changed to *daemon threads* using method **`setDaemon(boolean on)`**
  - If the only nonterminated threads are daemons, then the JVM will terminate
  - Daemon threads should only be used for "background work" (e.g. updating status bars, etc.) needed while "useful" computation is being performed
- **`setDaemon()`** is only valid if thread state is **`NEW;`** otherwise, `IllegalThreadStateException` thrown

Methods for Interacting with Scheduler

- `void setPriority(int newPriority)`
  Set priority to given value (must be between MIN_PRIORITY and MAX_PRIORITY: see below)
- `int getPriority()`
  Return priority value
- `static void yield()`
  "Hint" to scheduler that thread can give up processor
- `static void sleep(int millis)`
  Block for millis milliseconds
- `static int MIN_PRIORITY`
  Smallest (lowest) priority
- `static int MAX_PRIORITY`
  Largest (highest) priority
- `static int NORM_PRIORITY`
  Default priority

Thread Safety

- We assume that the scheduler can interleave or overlap threads arbitrarily
- Data can be shared across threads
- Can lead to *interference*
  - Storage corruption
  - Violation of representation invariant
  - Violation of a protocol (e.g., *A* occurs before *B*)

Thread Safety

- Programmer can have some influence via `yield()`, `setPriority()`, etc.

- But most decisions are outside user control, leading to possibilities for
  - Nondeterminism
  - *Interference*: threads overwrite each other's work

Example

```
public class Example extends Thread {
  private static int cnt = 0;  // shared state
  public void run() {
    int y = cnt;
    cnt = y + 1;
  }

  public static void main(String args[]) {
    Thread t1 = new Example();
    Thread t2 = new Example();
    t1.start();
    t2.start();
  }
}
```

Example: t1 finishes before t2

```
static int cnt = 0;
```

t1

```
run() {
  int y = cnt;
  cnt = y + 1;
}
```

t2

```
run() {
  int y = cnt;
  cnt = y + 1;
}
```

`cnt = 2;`

Example: t2 finishes before t1

```
static int cnt = 0;
```

t2

```
run() {
  int y = cnt;
  cnt = y + 1;
}
```

t1

```
run() {
  int y = cnt;
  cnt = y + 1;
}
```

cnt = 2;

Example: t1 and t2 interleave

```
static int cnt = 0;
```

t1

```
run() {
  int y = cnt; //y =0



  cnt = y + 1;//cnt=1

}
```

t2

```
run() {

  int y = cnt; // y = 0
  cnt = y + 1; //cnt=1


}
```

cnt = 1; why?

What Happened?

- The code read the counter value & then increments that value by one
- In the first example, **`t1`** was preempted after it read the counter but before it stored the new value.
- When **`t1`** resumed, it updated a stale value
- This is an example of a *data race*

Two Threads

```
public class TwoThreads {
  public static class Thread1 extends Thread {
    public void run() {
         System.out.println("A");
         System.out.println("B");
    }
}
public static class Thread2 extends Thread {
    public void run() {
                System.out.println("1");
                System.out.println("2");
    }
}
public static void main(String[] args) {
       new Thread1().start();
       new Thread2().start();
}}
```

Two Threads

```
public class TwoThreads {
  public static class Thread1 extends Thread {
    public void run() {
      System.out.println("A");
      System.out.println("B");
    }
}
public static class Thread2 extends Thread {
    public void run() {
      System.out.println("1");
      System.out.println("2");
    }
}
public static void main(String[] args) {
    new Thread1().start();
    new Thread2().start();
}}
```

Output:

12AB
1A2B
1AB2
A12B
A1B2
AB12