Tasks and threads

- Most applications are organized around the concept of *task*: abstract, discrete unit of work. Splitting the work into tasks simplifies program organization, facilitate error recovering, promotes concurrency
- Tasks are executed by threads (the mechanism)
- Sequential Web Server:

```
class SingleThreadWebServer {
        public static void main(String[] args) throws IOException {
            ServerSocket socket = new ServerSocket(80);
             while( true ){
                Socket connection = socket.accept(); //task == client request connection
                handleRequest( connection );
            }
         }
}//SingleThreadWebServer
```

Main thread receives and processes one single client request at a time. Performs poorly (in response time and throughput)

Creating a new thread per each task

```
class ThreadPerTaskWebServer {
        public static void main(String[] args) throws IOException {
            ServerSocket socket = new ServerSocket(80);
            while (true) {
                final Socket connection = socket.accept();
                Runnable task = new Runnable() {
                        public void run() {
                            handleRequest(connection);
                        }
                    };
                new Thread(task).start();
            }
        }
}//ThreadPerTaskWebServer
```

Tasks now are offloaded from main thread, execute in parallel, require task handling code to be thread-safe, may spawn a large number of threads which can consume a lot of resources (memory) and can make unstable the JVM

The Executor Framework

- Separates tasks specification from the mechanism (threads) used (according to a policy) for executing them. E.g. a thread pool
- The primary abstraction for task execution in Java libraries is Executor:

```
public interface Executor {
      void execute( Runnable command );
}
```

Task described by a Runnable object.
Using an executor is usually the easiest way to implement a producer-consumer application. The executor operates on the *task (work) queue*

Web server using a thread pool

```
class TaskExecutionWebServer {
        private static final int NTHREADS = 100;
        private static final Executor exec = Executors.newFixedThreadPool(NTHREADS);
        public static void main(String[] args) throws IOException {
            ServerSocket socket = new ServerSocket(80);
            while (true) {
                final Socket connection = socket.accept();
                Runnable task = new Runnable() {
                    public void run() {
                        handleRequest(connection);
                    }
                };
                exec.execute(task);
            }
        }//main
}//TaskExecutionWebServer
```

Executor that starts one thread per task

```
public class ThreadPerTaskExecutor implements Executor {
      public void execute(Runnable r) {
          new Thread(r).start();
      };
}
```

Executor that starts a task on the calling thread

```
public class WithinThreadExecutor implements Executor {
       public void execute(Runnable r) {
            r.run();
       };
}
```

Execution policies

- Are tied to decoupling task submissions from their execution. An execution policy specifies:
  - *In what thread will tasks be executed*?
  - *In what order should tasks be executed* (FIFO, LIFO, priority order)?
  - *How many tasks may execute concurrently*?
  - *How many tasks may be queued pending execution*?
  - *If a task has to be rejected because the system is overloaded, which task should be selected as the victim, and how should the application be notified?*
  - *What actions should be taken before or after executing a task?*

Thread pools – common executors built by factory methods of utility class Executors

newFixedThreadPool. A fixed-size thread pool creates threads as tasks are submitted, up to the maximum pool size, and then attempts to keep the pool size constant (adding new threads if a thread dies due to an unexpected Exception).

newCachedThreadPool. A cached thread pool has more flexibility to reap idle threads when the current size of the pool exceeds the demand for processing, and to add new threads when demand increases, but places no bounds on the size of the pool.

newSingleThreadExecutor. A single-threaded executor creates a single worker thread to process tasks, replacing it if it dies unexpectedly. Tasks are guaranteed to be processed sequentially according to the order imposed by the task queue (FIFO, LIFO,priority order).

Single-threaded executors also provide sufficient internal synchronization to guarantee that any memory writes made by tasks are visible to subsequent tasks; this means that objects can be safely confined to the "task thread“ even though that thread may be replaced with another from time to time.

newScheduledThreadPool. A fixed-size thread pool that supports delayed and periodic task execution, similar to Timer.

ExecutorService interface

- Extends Executor and provides methods for controlling the lifecycle of an executor (whose states are: *running*, *shutting down*, *terminated*) and to ri-express submission

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit)throws InterruptedException;
     // ... additional convenience methods for task submission
}
```

ScheduledThreadPoolExecutor

- It is a valuable replacement of java.util.Timer service. Instead of one single “vulnerable” thread, multiple threads can be devoted to task scheduling and execution
- Allows tasks to be scheduled for delayed and/or repeated (periodic) execution

DelayQueue

•Is a BlockingQueue implementation that provides functionality of ScheduledThreadPoolExecutor

•Manages a queue of Delayed objects, each associated with a delay time.

•DelayQueue permits to take elements, according to their delay time, but only when the relevant delay was expired.

Callable and Future

- A **Callable<V>** is a “Runnable” which can return a result of type V (Void for no result)
- The task of a callable is programmed in its V call() method
- Both Runnable and Callable are tasks whose execution is asynchronous
- A **Future<V>** is an object useful to check about termination, getting the result, cancelling a Callable
- The *submit* methods of an Executor returns a Future object
- The **FutureTask** is a concrete class whose objects are both tasks (Runnable or Callable) and Future. Of course, a future task can be submitted to an executor, or it can be directly executed by invoking its run() method

Example

```
Callable<Integer> myComputation = . . .;
FutureTask<Integer> task =
      new FutureTask<Integer>(myComputation);
Thread t = new Thread(task); // it's a Runnable
t.start();
. . .
Integer result = task.get(); // it's a Future
```

java.util.concurrent.Callable<V> - Java 5 or up

- **V call()**

runs a task that yields a result.

java.util.concurrent.Future<V> - Java 5 or up

- **V get()**
- **V get(long time, TimeUnit unit)**

gets the result, blocking until it is available or the given time has elapsed. The second method throws a TimeoutException if it was unsuccessful.

- **boolean cancel(boolean mayInterrupt)**

attempts to cancel the execution of this task. If the task has already started and the mayInterrupt parameter is true, it is interrupted. Returns true if the cancellation was successful.

- **boolean isCancelled()**

returns true if the task was canceled before it completed.

- **boolean isDone()**

returns true if the task completed, through normal completion, cancellation, or an exception.

java.util.concurrent.FutureTask<V> - Java 5 or up

- **FutureTask(Callable<V> task)**
- **FutureTask(Runnable task, V result)**

constructs an object that is both a Future<V> and a Runnable.

java.util.concurrent.Executors - Java 5 or up

- **ExecutorService newCachedThreadPool()**

returns a cached thread pool that creates threads as needed and terminates threads that have been idle for 60 seconds.

- **ExecutorService newFixedThreadPool(int threads)**

returns a thread pool that uses the given number of threads to execute tasks.

- **ExecutorService newSingleThreadExecutor()**

returns an executor that executes tasks sequentially in a single thread.

- **ScheduledExecutorService newScheduledThreadPool(int threads)**

returns a thread pool that uses the given number of threads to schedule tasks.

- **ScheduledExecutorService newSingleThreadScheduledExecutor()**

returns an executor that schedules tasks in a single thread.

java.util.concurrent.ExecutorService - Java 5 or up

- **Future<T> submit(Callable<T> task)**
- **Future<T> submit(Runnable task, T result)**
- **Future<?> submit(Runnable task)**

submits the given task for execution.

- **void shutdown()**

shuts down the service, completing the already submitted tasks but not accepting new submissions.

java.util.concurrent.ScheduledExecutorService - Java 5 or up

- **ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)**
- **ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)**

schedules the given task after the given time has elapsed.

- **ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)**

schedules the given task to run periodially, every period units, after the initial delay has elapsed.

- **ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)**

schedules the given task to run periodially, with delay units between completion of one invocation and the start of the next, after the initial delay has elapsed.

Controlling group of tasks: java.util.concurrent.ExecutorCompletionService

- **ExecutorCompletionService(Executor e)**

constructs an executor completion service that collects the results of the given executor.

- **Future<T> submit(Callable<T> task)**
- **Future<T> submit(Runnable task, T result)**

submits a task to the underlying executor.

- **Future<T> take()**

removes the next completed result, blocking if no completed results are available.

- **Future<T> poll()**
- **Future<T> poll(long time, TimeUnit unit)**

removes the next completed result or null if no completed results are available.The second method waits for the given time.

Parallel File Searching

- The goal is to input a certain number of text file names, and a keyword, and find how many files contains the given keyword
- An obvious sequential solution consists in opening one file at time, search for the existence of the keyword, counting the search, close the file and continuing with the next file
- In the following a concurrent solution is shown where each file is independently searched by a distinct task. Tasks are then managed by a thread pool

```
package concur.threadpool;
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
public class ParallelSearchFile {
      private static class Searcher implements Callable<Boolean>{
            private final File file;
            private String keyword;
            public Searcher( final File file, String keyword ){
                  this.file=file; this.keyword=keyword;
            }
            public Boolean call(){
                  try{
                        Scanner sc=new Scanner( file );
                        boolean trovato=false;
                        while( !trovato && sc.hasNextLine() ){
                              String linea=sc.nextLine();
                              if( linea.contains(keyword) ) trovato=true;
                        }
                        sc.close();
                        return trovato;
                  }catch(IOException e){
                        e.printStackTrace();
                        return false;
                  }
            }//call
}//Searcher
```

```
public static void main( String []args ) throws InterruptedException{
      if( args.length<=1 ){
          System.out.println("Argomenti attesi: file1 file2 ... fineN chiave"); System.exit(-1);
       }
       File []file=new File[args.length-1];
       for( int i=0; i<args.length-1; i++ ) file[i]=new File( args[i] );
       String chiave=args[args.length-1];
       ArrayList<Future<Boolean>> risultati = new ArrayList<Future<Boolean>>();
       ExecutorService pool=Executors.newFixedThreadPool(10);
       for( int i=0; i<file.length; i++ ){
            Callable<Boolean> callable=new Searcher( file[i], chiave );
            Future<Boolean> f=pool.submit( callable );
            risultati.add( f );
       }
       //waiting and counting positive results
       int conta=0;
       for( Future<Boolean> f: risultati )
             try{
                  if( f.get() ) conta++;
             }catch( ExecutionException e ){}

       System.out.println("Numero di file contenenti \""+chiave+"\" = "+conta );
       pool.shutdown();
}//main
```

Using CompletionService

- Previous implementation builds worker tasks, submits them to the thread pool executor, achieves the Future objects, finally waits for each worker to terminate so as to check its boolean result
- Doing the for-each on the result forces but unnecesarily to wait workers in the order of their submission. It could be more flexible to wait for worker termination in the true asynchronous way this happens
- It is then useful to reformulate the solution using the services of CompletionService. Only the new main() is shown (the Callable worker remains unchanged)
- CompletionService *decorates* (in the sense of Decorator pattern) an executor. It understands the same operation and exposes some new ones. It hosts a BlockingQueue for delivering terminating tasks

```
public static void main( String []args ) throws InterruptedException{
    if( args.length<=1 ){
       System.out.println("Argomenti attesi: file1 file2 ... fineN chiave"); System.exit(-1);
    }
    File []file=new File[args.length-1];
    for( int i=0; i<args.length-1; i++ ) file[i]=new File( args[i] );
    String chiave=args[args.length-1];
    ExecutorService pool=Executors.newFixedThreadPool(10);
    CompletionService<Boolean> completionService =
          new ExecutorCompletionService<Boolean>( pool );
    for( int i=0; i<file.length; i++ ){
       completionService.submit( new Searcher( file[i], chiave ) );
    }

    int conta=0;
    for( int i=0; i<args.length-1; ++i ){ //here we only now how many tasks were submitted
       try{
          Future<Boolean> f = completionService.take(); //any one tasks terminated
          if( f.get() ) conta++;
       }catch( ExecutionException e ){}
     }

    System.out.println("Numero di file contenenti \""+chiave+"\" = "+conta );
    pool.shutdown();
}//main
```

Parallelizing recursive algorithms

```
void processSequentially(List<Element> elements) {
      for (Element e : elements)
            process(e);
}//processSequentially
```

Provided the various “processes” are independent one to another, e.g. the use distinct data, the overall computation can be easily parallelized as follows:

```
void processInParallel(Executor exec, List<Element> elements) {
      for (final Element e : elements)
            exec.execute(new Runnable() {
                  public void run() { process(e); }
            });
}//processInParallel
```

Transforming sequential tail-recursion into parallelized recursion

Depth-first traversal of a tree, performing some computation on each node, and putting the result in a collection

```
public<T> void sequentialRecursive(List<Node<T>> nodes, Collection<T> results) {
        for (Node<T> n : nodes) { //for each node
              results.add(n.compute()); //process the node
              sequentialRecursive(n.getChildren(), results); //activate the process on children of this node
        }
}//sequentialRecursive

public<T> void parallelRecursive(final Executor exec, List<Node<T>> nodes, final Collection<T> results) {
       for (final Node<T> n : nodes) { //for each node
             exec.execute(new Runnable() { //spawn a task for processing the node
                    public void run() {
                          results.add(n.compute());
                    }
             });
            parallelRecursive(exec, n.getChildren(), results); //activate process on children
        }
}//parallelRecursive
```

Waiting for results to be calculated in parallel

```
public<T> Collection<T> getParallelResults( List<Node<T>> nodes )
    throws InterruptedException {
    ExecutorService exec = Executors.newCachedThreadPool();
    Queue<T> resultQueue = new ConcurrentLinkedQueue<T>();
    parallelRecursive( exec, nodes, resultQueue );
    //executor shutdown
    exec.shutdown();
    exec.awaitTermination( Long.MAX_VALUE, TimeUnit.SECONDS );
    return resultQueue;
}//getParallelResults
```

A Parallel Merge Sort

- This well-known recursive algorithm, based on divide-and-conquer strategy, can be easily parallelized by assigning a different task to each recursive call on a sub segment of the data array
- Each task operates on distinct data elements
- Before merging two parallel sorted segments, it is necessary to wait for their termination

```
package concur.mergesort;
import java.util.concurrent.*;
import java.util.*;

public class ParallelMergeSort {
    private final ExecutorService exec = Executors.newCachedThreadPool();

    private class SorterTask implements Runnable{
        private final int[] a;
        private final int inf, sup;
        public SorterTask( final int[] a, final int inf, final int sup ){
            this.a=a; this.inf=inf; this.sup=sup;
    }
    public void run(){
        if( inf<sup ){
            int med = (inf+sup)/2;
            Future<?> f1 = exec.submit( new SorterTask(a,inf,med) );
            Future<?> f2 = exec.submit( new SorterTask(a,med+1,sup) );
            try{
                f1.get(); f2.get();
            }catch( Exception e ){
                e.printStackTrace();System.exit(-1);
            }
            merge(a,inf,med,sup);
        }
    }//run
```

```
        public void merge( int[] a, int inf, int med, int sup ){
                int[] aux = new int[sup-inf+1];
                int i=inf, j=med+1, k=0;
                while( i<=med && j<=sup ){
                        if( a[i]<=a[j] ){ aux[k]=a[i]; ++i; ++k; }
                        else{ aux[k]=a[j]; ++j; ++k; }
                }
                while( i<=med ){
                        aux[k]=a[i]; ++i; ++k;
                }
                while( j<=sup ) {
                        aux[k]=a[j]; ++j; ++k;
                }
                for( k=0; k<aux.length; ++k ) a[inf+k]=aux[k];
        }//merge
}//SorterTask

public ParallelMergeSort( int[] a ) throws Exception{
        Future<?> f = exec.submit( new SorterTask( a, 0, a.length-1 ) );
        f.get();
        exec.shutdown();
        exec.awaitTermination( Long.MAX_VALUE, TimeUnit.SECONDS );
}//parallelMergeSort
```

```
    public static void main( String[] args ) throws Exception{
        int a[]={ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
        System.out.println( Arrays.toString(a) );
        new ParallelMergeSort( a );
        System.out.println( Arrays.toString(a) );
    }//main

}//ParallelMergeSort
```