Practical Concurrent and Parallel Programming V

IT University of Copenhagen

Friday 2020-09-25

Kasper Østerbye

Starts at 8:00

Plan for today

Follow up on last week

Executor framework

Follow up and loose ends

- Thanks for rapid feedback on missing files

Follow up and loose ends

- Thanks for rapid feedback on missing files

Outliers

- No firm definition (or rather there are many) of when a measurement is an outlier
- They are often the source of standard deviation becoming large or flucturating
- **Systematic** outliers should be investigated - they point to something you have overlooked
- JIT compilation can cause outliers

JIT compiler I

```
private static long incr1000(long n){
  long sum = 0;
  for (int i = 0; i<1000; i++)
    sum += incr10(i);
  return sum;
}
private static long incr10(long n){
  long cnt = 0;
  for (int i = 0; i<10; i++)
    cnt++;
  return cnt;
}
```

JIT compiler II

```
static final String fs = " - n: %d bestTime: %,d thisTime: %,d\n";
public static void main(String[] args) {
  SystemInfo();
  boolean printNext = false;
  long bestTime = Long.MAX_VALUE;
  for(long n=1; n<100_000; n++){
    long start = System.nanoTime();
    for (long i = 0; i < n; i++)
      incr1000(i);
    long time = (long)(System.nanoTime()-start)/n;

    if (printNext){
      System.out.format("NEXT"+fs, n, bestTime, time);
      printNext = false;
    }
    if (time > bestTime*3){ // Outlier - maybe JIT
      System.out.format("SLOW"+fs, n, bestTime, time);
      printNext = true;
    }
    if (time < bestTime) bestTime = time;
    if (time == 0) {
      System.out.format("LAST"+fs, n, bestTime, time);
      return;
    }
  }
}
```

JIT compiler III

```
$ java -Xbatch MeasureJIT
# OS:   Mac OS X; 10.15.6; x86_64
# JVM:  Oracle Corporation; 1.8.0_212
# CPU:  null; 12 "cores"
# Date: 2020-09-24T20:47:47+0200
SLOW - n: 11 bestTime: 15,255 thisTime: 116,390
NEXT - n: 12 bestTime: 15,255 thisTime: 3,266
SLOW - n: 14 bestTime: 3,266 thisTime: 124,637
NEXT - n: 15 bestTime: 3,266 thisTime: 61
SLOW - n: 44 bestTime: 36 thisTime: 111
NEXT - n: 45 bestTime: 36 thisTime: 42
SLOW - n: 112 bestTime: 36 thisTime: 14,128
NEXT - n: 113 bestTime: 36 thisTime: 38
SLOW - n: 347 bestTime: 35 thisTime: 3,360
NEXT - n: 348 bestTime: 35 thisTime: 24
SLOW - n: 449 bestTime: 23 thisTime: 16,372
NEXT - n: 450 bestTime: 23 thisTime: 36
LAST - n: 453 bestTime: 0 thisTime: 0
```

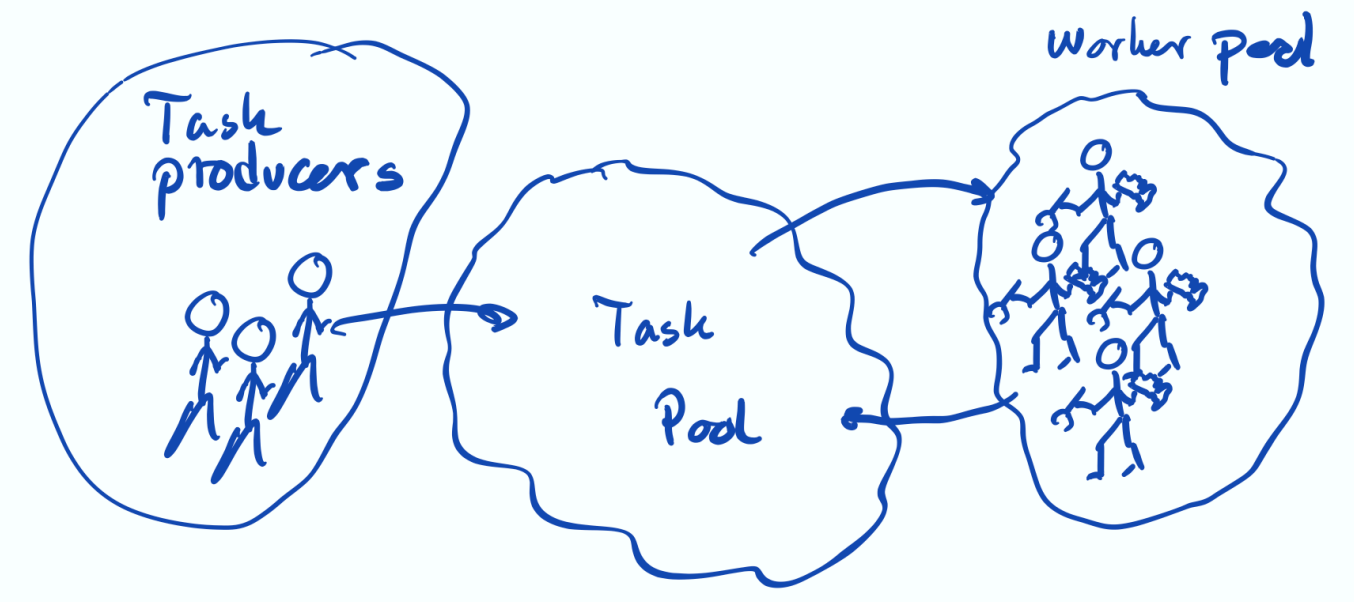
Normally the JIT compiler runs in a background thread. The `-Xbatch` flag tells the JVM to use the main thread for the JIT compiler.

Executor framework in Java

Motivation

- Threads are expensive to start - can we reuse a thread.
- Problem heap - breaking a problem down to smaller tasks

Executor framework - basics



(I like stick-men as threads, as that remind me threads are active)

Two kinds of threads: *Task producers*, and *Workers* (who do the tasks).

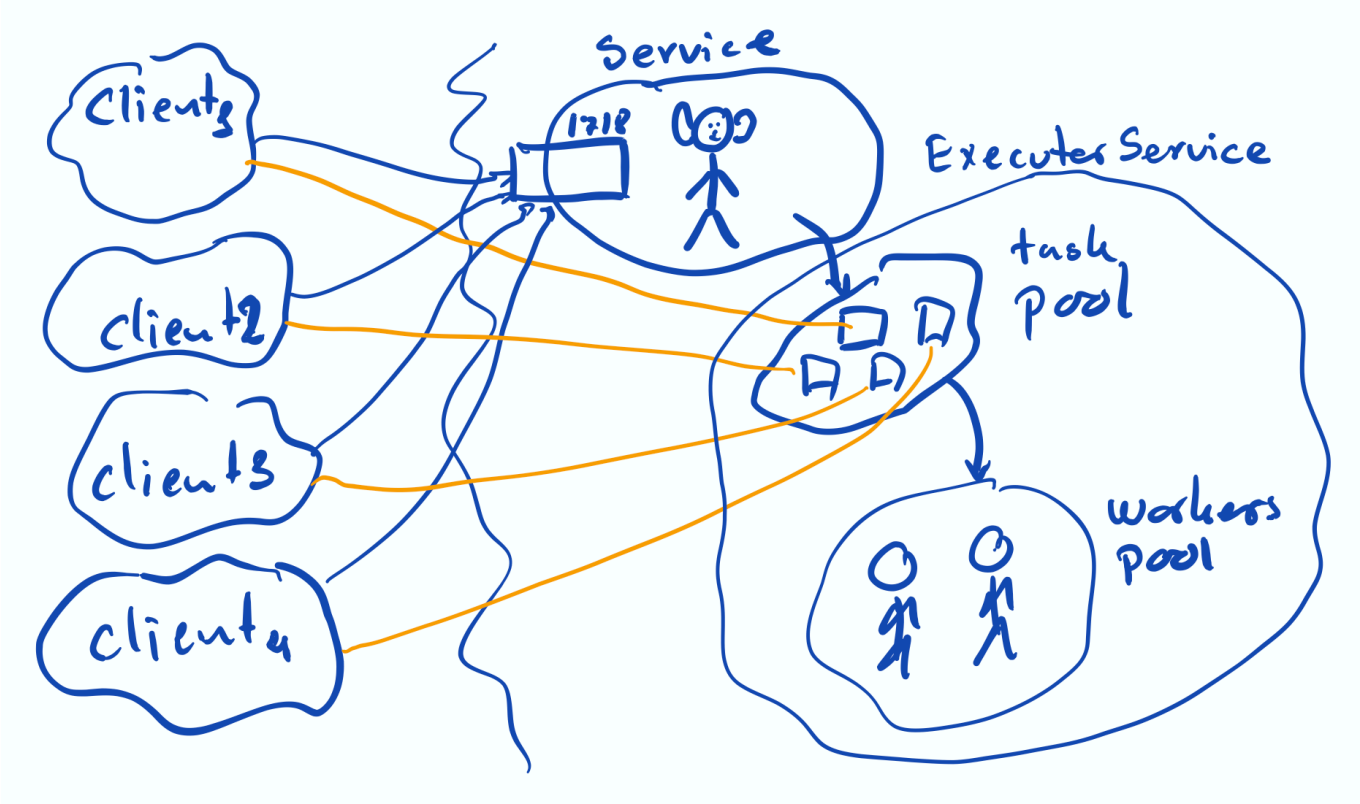
One (*important*) possibility is that the workers produce new tasks in the process of working on a task.

The Task pool and Worker pool together is called an *Executor service*.

Example

```
class NetworkService {
  private final ServerSocket serverSocket;
  private final ExecutorService pool;

  public NetworkService(int port, int poolSize) throws IOException {
    serverSocket = new ServerSocket(port);
    pool = Executors.newFixedThreadPool(poolSize);
  }

  public void runService() throws Exception { // run the service
    new Thread ( () -> {
      while (true)) {
        Socket clientConnection = serverSocket.accept(); // Blocking
        pool.execute(new Handler( clientConnection ));
      }}).start();
  }
}

class Handler implements Runnable {
  private final Socket socket;
  Handler(Socket socket) { this.socket = socket; }
  public void run() {
    // WORK: read and service request on socket
  }
}
```

Example explained I



Example explained II

The NetworkService is listing (big ears) on port 1718. The listening for client connections is the call `serverSocket.accept()`.

- The Service thread is blocked by that call.
- The Service thread is only listening when in that call.

When a client calls, a new Handler is created and added to the ExecutorService `pool`. The Handler is "the task".

Worker poolsize is two. When there is work, one of the worker threads can pick up the Handler/Task and execute the run method of the handler.

- This might be communicating back and forth with the client over the client connection (orange connection).
- It is the worker who should terminate the connection (or the client)
- The worker than then pick a new handler/task from the task pool

Trivial Executor services

(For illustration)

Run in task-submitters thread:

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

Run all in a new task:

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

The interface `Executor` is an early version (and superclass) of `ExecutorService`.

What is a task?

Two choices:

- Runnable - A piece of work where procucer do not need a result
- Callable/Future - A piece of work where the producer need the result

In the example above, it is a runnable passed to the executor service (`pool`).

The only method of `Executor` is `execute()`:

Modifier and Type	Method and Description
`void`	**`execute`**`(`**`Runnable`** `command)` Executes the given command at some time in the future.

Tasks with results

The sub-interface of `Executor` - `ExecutorService` adds a number of `submit()` methods in addition to `execute()`.

`<T> Future<T>`	**`submit(Callable<T> task)`** Submits a value-returning task for execution and returns a Future representing the pending results of the task.
`Future<?>`	**`submit(Runnable task)`** Submits a Runnable task for execution and returns a Future representing that task.
`<T> Future<T>`	**`submit(Runnable task, T result)`** Submits a Runnable task for execution and returns a Future representing that task.

javadoc ExecutorService

< T > and < ? >

Java type parameters are used in primarily two situations:

- Making collections libraries which are type specific - HashMap
- Generic functions - which is what we use with `ExecutorService` and `Future`

```
<T> Future<T>          submit(Callable<T> task)
                       Submits a value-returning task for execution and returns a Future
                       representing the pending results of the task.
```

```
@FunctionalInterface
public interface Callable<V>
```

A task that returns a result and may throw an exception. Implementors define a single method with no arguments called call.

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception. (*Javadoc*)

Future

Javadoc:

https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Future.html

```
interface ArchiveSearcher { String search(String target); }

class App {
  ExecutorService executor = ...
  ArchiveSearcher searcher = ...
  void showSearch(String target) throws InterruptedException {
    Callable<String> task = () -> searcher.search(target);
    Future<String> future = executor.submit(task);
    displayOtherThings(); // do other things while searching
    try {
      displayText(future.get()); // use future
    } catch (ExecutionException ex) { cleanup(); return; }
  }
}
```

And what about the <?>

Again - javadoc to your rescue:

https://docs.oracle.com/javase/tutorial/java/generics/index.html

The <?> is an *unbounded wildcard*

> "There are two scenarios where an unbounded wildcard is a useful approach:
>
> - If you are writing a method that can be implemented using functionality provided in the Object class.
> - When the code is using methods in the generic class that don't depend on the type parameter. "

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

And what about the <?>

Again - javadoc to your rescue:

https://docs.oracle.com/javase/tutorial/java/generics/index.html

The <?> is an *unbounded wildcard*

> "There are two scenarios where an unbounded wildcard is a useful approach:
>
> - If you are writing a method that can be implemented using functionality provided in the Object class.
> - When the code is using methods in the generic class that don't depend on the type parameter. "

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Try javadoc before stackoverflow - at least occasionally.

A mystery

From javadoc of `ExecutorService`:

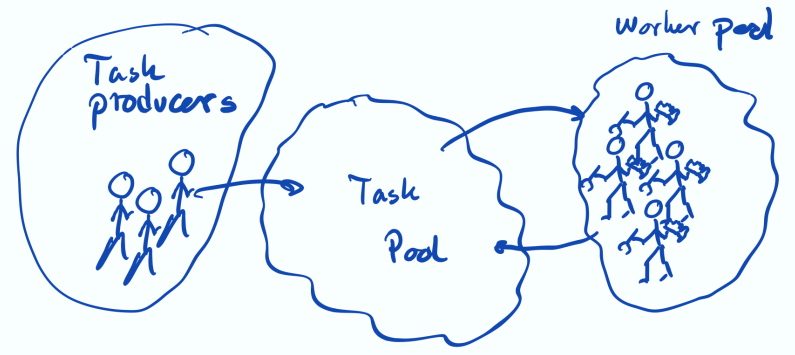
```
<T> Future<T> submit(Runnable task, T result)
```

> Submits a Runnable task for execution and returns a Future representing that task. The Future's get method will return the given result upon successful completion.

Why would one pass the result to be returned by running the task????

Execution policies

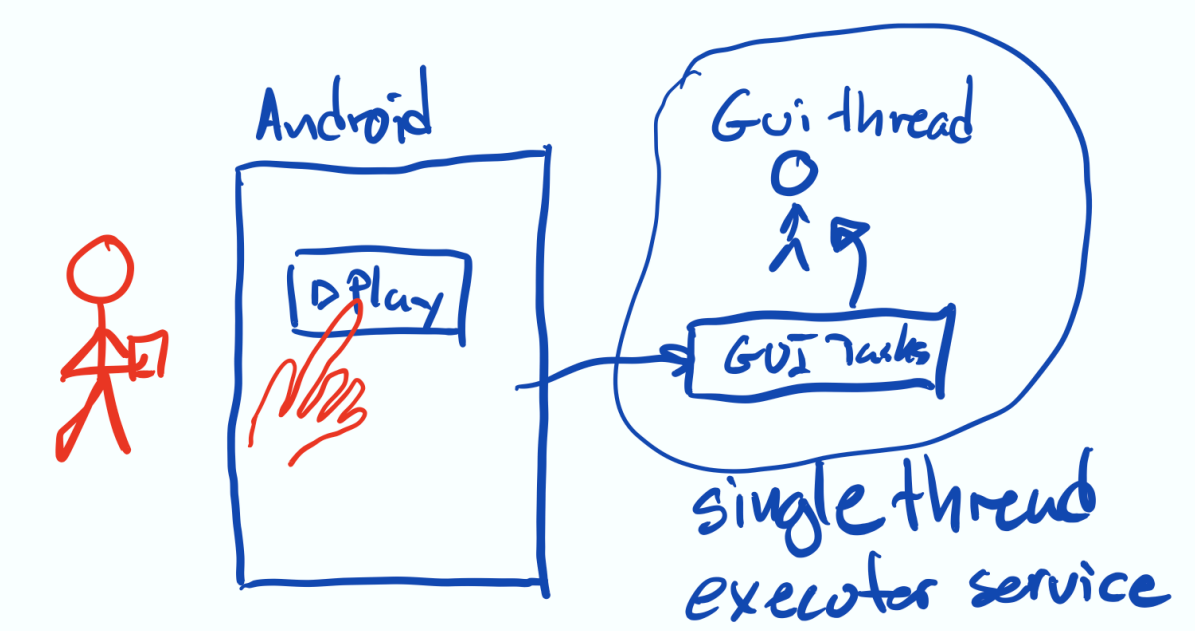
A lot of the design choices relate to the two pools



- 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/task producer be notified?
- What actions should be taken before or after executing a task?

Android GUI-thread

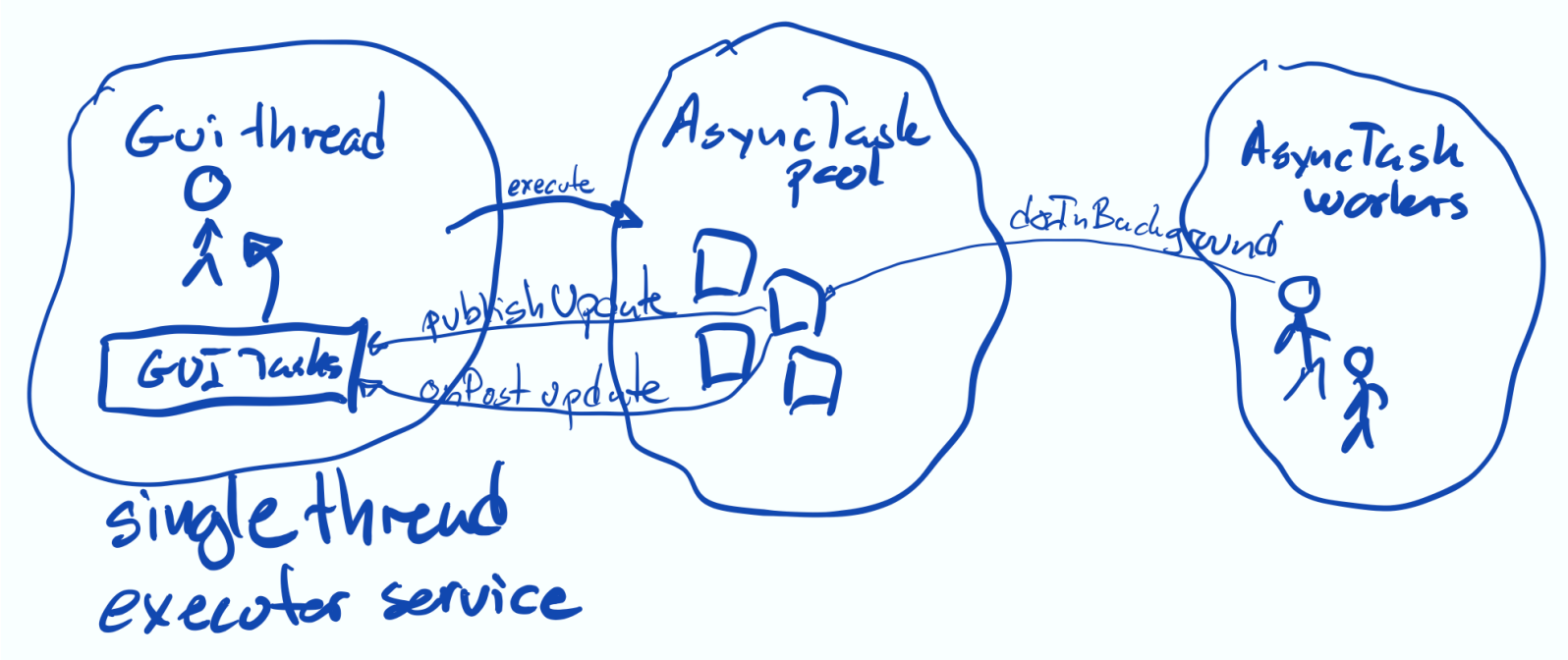
```
final Button button = findViewById(R.id.button_id);

button.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
    // Code here executes on main thread after user presses button
  }
});
```



Android AsyncTask

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

AsyncTask picture



The two methods `publishProgress` and `onPostExecute` puts tasks into the task-pool of the GUI thread.

The GUI thread is a prime example of a single thread executor service.

Task size

One large task can monopolize one of the worker threads.

Sometimes a task should have as part of its logic to break itself down into smaller pieces.

Task size

One large task can monopolize one of the worker threads.

Sometimes a task should have as part of its logic to break itself down into smaller pieces.

More comments please

At LearnIT under Week5 there is a poll. All feedback is valuable for us.

Thread pools

There are different structures of thread pools build in as factory methods of `Executors`

- newFixedThreadPool. A fixed-size thread pool.
- newCachedThreadPool. variable size, places no bounds on the size of the pool.
- newSingleThreadExecutor. Memory writes made by tasks are visible to subsequent tasks.
- newScheduledThreadPool. A fixed-size thread pool that supports delayed and periodic tasks.

Thread pools II

I would like to advocate that one of your primary sources is:
https://docs.oracle.com/javase/8/docs/api/

An other source (literally) is:
http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/

The source for `Executors` show that the factory methods delegate to `ThreadPoolExecutor`.

Work stealing thread pool

Motivation

Strategy

Example - compute total size of directory

Stolen from: https://javacreed.com/java-fork-join-example

```
class DirSizeSeq {
  public static long sizeOf(final File file) {
    long size = 0;

    if (file.isFile()) {
      size = file.length();
    } else {

      final File[] children = file.listFiles();
      if (children != null) {

        for (final File child : children) {
          size += DirSizeSeq.sizeOf(child);
        }

      }
    }
    return size;
  }
}
```

As ForkJoinPool

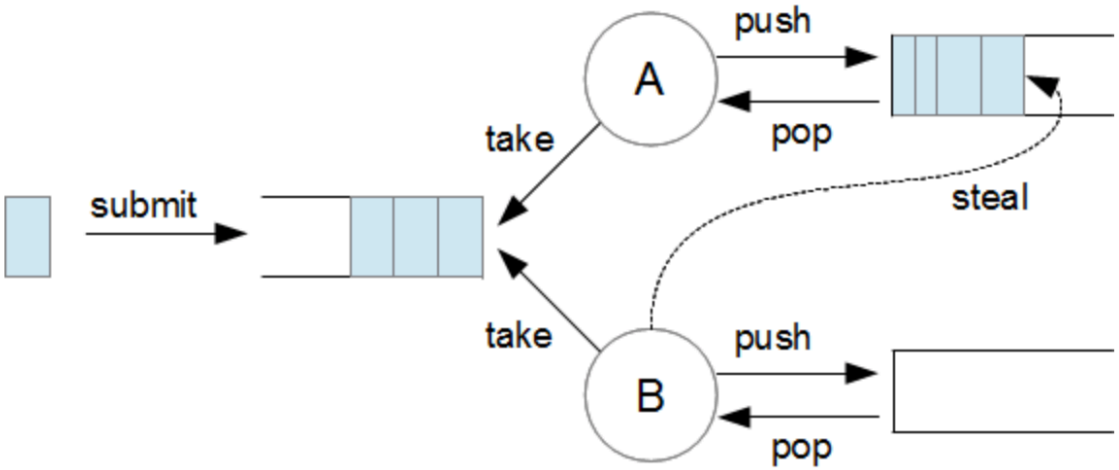
```
class DirSizeTask extends RecursiveTask<Long> {
  // ...
  protected Long compute() {
    if (file.isFile()) {
      return file.length();
    }

    final List<DirSizeTask> tasks = new ArrayList<>();
    final File[] children = file.listFiles();
    if (children != null) {
      for (final File child : children) {
        final DirSizeTask task = new DirSizeTask(child);
        task.fork();
        tasks.add(task);
      }
    }

    long size = 0;
    for (final DirSizeTask task : tasks) {
      size += task.join();
    }

    return size;
  }
}
```

Code continued

```
public static long sizeOf(final File file) {
  final ForkJoinPool pool = new ForkJoinPool();
  try {
    return pool.invoke(new DirSizeTask(file));
  } finally {
    pool.shutdown();
  }
}
```

Work stealing

Stolen from: [The fork/join framework in Java 7]



A ForkJoinPool with two worker threads, A and B. In addition to the inbound queue, each thread also has its own internal task queue. Each thread starts by processing its own queue, before moving on to other queues (figure 1).

To avoid contention, each worker puts its own forks on its own queue.

1. Take from own queue
2. Steal from other or
3. Take from shared

Shut down

The `ExecutorService` has methods to help in shutting down.

```
try {
  while (true) {
    pool.execute(new Handler(serverSocket.accept()));
  }
} catch (IOException ex) {
  pool.shutdown();
}
```

The challenge is **when** to shut down.

- Perhaps the problem creator knows (all input read, all results obtained, or something similar).
- Perhaps user asked to close app
- Perhaps no new tasks were added for some time?

The javadoc for `ExecutorService` has the list of methods to support orderly shutdown, as well as some examples.

Executor framework in .Net

It is called Task Parallel Library (TPL).

It is not quite the same as many things in .Net uses `async`. In many ways .Net is one generation ahead of Java wrt. concurrency.

We will later in the class look at Kotlin, which is again one generation ahead of .Net.

Outlook

- How to handle large tasks? How to make a task pause and release the thread to an other task?

Outlook

- How to handle large tasks? How to make a task pause and release the thread to an other task?

This class was deprecated in API level 30.

Use the standard `java.util.concurrent` or Kotlin concurrency utilities instead.

We will look at Kotlin concurrency and coroutines in particular later in the class (one of the completely new aspects of this course).