Multicore Programming
Future Composition

Louis-Claude Canon
louis-claude.canon@univ-fcomte.fr

Bureau 414C

Master 1 computer science – Semester 8
Motivation

- Express multitask operations in a declarative way without writing the thread logic.
- Rely on *future composition* to limit synchronization code (since Java 8).
Outline

Blocking Problem

Future Compositon

Summary and References
Outline

Blocking Problem
With Thread Pools
With Future

Future Compositon

Summary and References
Context

- Tasks (or user threads) cannot be preempted.
- A (system) thread must finish a task before proceeding to another task.
- When a task is waiting for a blocking call, its thread is idle even though other tasks are ready.
- *Blocking/non-blocking* is also referred to as *synchronous/asynchronous*. 
Queued tasks

Five worker threads running five active tasks
Sizing Thread Pools: Problem

- A large number of tasks is submitted to a thread pool.
- Each task spends 75% of the time in a blocking operation.
- There are 2 cores.
- What is the minimum number of threads required in the pool to prevent any core from being idle?
Sizing Thread Pools: Solution

- In the best case, the periods for all blocking operations are well distributed (overlapping is minimum).
- The minimum number of necessary threads is 8: at any point in time, 75% of them (six) execute blocking tasks, while all cores are busy executing the remaining two.
- When tasks are waiting 90% of the time, 20 threads are necessary.
Gantt Chart with Blocking (2 threads)

Thread pool

$P_1$  

$P_2$

0 1 2 3 4  

time
Gantt Chart with Blocking (4 threads)

Thread pool

$P_2$

$P_1$

0 1 2 3 4 time
Gantt Chart without Blocking (8 threads)

Thread pool

$P_1$:

$P_2$:

time

0 1 2 3 4

0 1 2 3 4

Louis-Claude Canon

MCP – Future Composition
Threading Models

Kernel (or system) threads vs. user/application threads:

1:1  kernel-level threading (preemptive multitasking): native threads
n:1  user-level threading (cooperative multitasking)
m:n  hybrid threading: thread pool, fiber (shared memory), green threads, virtual threads, etc.
Cooperative Multitasking

- The objective is to make each core busy with the minimum number of threads.
- Cooperation between tasks is needed to avoid blocking threads as much as possible.
- Solution 1 (Java): each task should avoid blocking operations.
- Solution 2 (Erlang): each task should frequently yield execution to a user scheduler and resumes it later (continuation-style).
Outline

Blocking Problem
  With Thread Pools
  With Future

Future Composition

Summary and References
Asynchronous Operations

Asynchronous calls with future limit blocking and improve clarity:
▶ Launch multiple tasks as soon as possible to limit the time spent blocking while they complete.
▶ Objectives of Future:
  ▶ improve code clarity
  ▶ minimize blocking with asynchronous operations
  ▶ limit the number of threads (using executors)
Future Principle

Blocking Problem With Future
Overcoming Future Limitation

- No way to specify how to react to a future completion without a blocking thread.
- Future *composition* allows minimizing blocking even more by chaining asynchronous operations and specifying how computations should proceed asynchronously.
- Eliminate the need of a supervising blocking thread.
Outline

Blocking Problem

Future Composition
  Basic Interface and Example
  Extended Interface

Summary and References
CompletionStage

Many available chaining methods:

```java
stage.thenApply(x -> square(x))
    .thenAccept(x -> System.out.print(x))
    .thenRun(() -> System.out.println());
```

Complete the stage, then compute a square root, then print it, then print an empty line.
Execution Diagram

\[ \sqrt{x} \rightarrow \text{print}(x) \rightarrow \text{println} \]

thread \hspace{1cm} thread

\[ \text{then.} \]

\[ \times \]

\[ \text{square}(x) \rightarrow \text{print}(x) \rightarrow \text{println}() \]
Creation of CompletableFuture

▶ Implement Future and CompletionStage.
▶ Creation with a task (submitted to the default ForkJoinPool):

```java
static CompletableFuture<Void> runAsync(Runnable runnable)
static CompletableFuture<U> supplyAsync(Supplier<U> supplier)
```

▶ Creation by combining multiple CompletableFuture:

```java
static CompletableFuture<Void>
    allOf(CompletableFuture<?>... cfs)
static CompletableFuture<Object>
    anyOf(CompletableFuture<?>... cfs)
```
Creation of CompletableFuture

```
thread

supplyAsync

thread

computation
```
Creation of CompletableFuture

```
allOf(cf1, cf2, ..., cfn)
```
Completion of CompletableFuture

- Basic completions:

```java
boolean complete(T value)
T join()
```

- The name CompletableFuture comes from the method `complete`.

- Completion with a timeout:

```java
CompletableFuture<T> completeOnTimeout(T value,
             long timeout, TimeUnit unit)
CompletableFuture<T> orTimeout(long timeout,
             TimeUnit unit)
```
Parallel Streams vs. Futures

Assume a request needs to be processed with different arguments:

```java
List<Result> result = arguments.stream()
    .map(arg -> f(arg))
    .toList();
```

Two parallelization options: using a parallel stream, or asynchronous operations.
Parallelizing with Futures

```java
List<CompletableFuture<Result>> comps = arguments.stream()
    .map(arg -> CompletableFuture.supplyAsync(() -> f(arg)))
    .toList();

List<Result> result = comps.stream()
    .map(CompletableFuture::join)
    .toList();
```

Must be in two separate streams due to lazy evaluation (otherwise, there is no parallelism).
Parallelizing with Futures (Alternative)

If there is no result:

```java
arguments.stream()
    .map(arg -> CompletableFuture.runAsync(() -> f(arg)))
    .reduce(CompletableFuture::allOf)
    .get()
    .join();
```

Favor parallel streams for CPU-intensive tasks and streams of futures with I/O bound tasks.
Avoiding Blocking

```java
int y = f(x);
int z = g(x);
System.out.println(y + z);
CompletableFuture<Integer> c1 = new CompletableFuture<>();
CompletableFuture<Integer> c2 = new CompletableFuture<>();
c1.thenCombine(c2, (y, z) -> System.out.println(y + z));
executor.submit(() -> c1.complete(f(x)));
executor.submit(() -> c2.complete(g(x)));```

```java
```
Execution Diagram

```
f(x)
```

```
g(x)
```

```
println
```

```
thread
```

```
thread
```

```
thread
```

```
submit
```

```
submit
```

```
submit
```

```
println
```

```
f(x)
```

```
g(x)
```

```
println
```
Outline

Blocking Problem

Future Composition
  Basic Interface and Example
  Extended Interface

Summary and References
Sequence of CompletionStage<T>

Compl.Stage<Void> thenRun (Runnable)
Compl.Stage<Void> thenAccept (Consumer<T>)
Compl.Stage<U> thenApply (Function<T,U>)
Compl.Stage<U> thenCompose(Func.<T,Compl.Stage<U>>)
Composition of CompletionStage<T>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>runAfterEither</code></td>
<td>Runs a Runnable after the stage completes an event.</td>
</tr>
<tr>
<td><code>acceptEither</code></td>
<td>Accepts an event and a Consumer.</td>
</tr>
<tr>
<td><code>applyToEither</code></td>
<td>Applies a function to an event.</td>
</tr>
<tr>
<td><code>runAfterBoth</code></td>
<td>Runs a Runnable after the stage completes both events.</td>
</tr>
<tr>
<td><code>thenAcceptBoth</code></td>
<td>Accepts both events and a BiConsumer.</td>
</tr>
<tr>
<td><code>thenCombine</code></td>
<td>Combines two events with a BiFunction.</td>
</tr>
</tbody>
</table>
Methods Nomenclature

- discard “then”, “after” and “to”
- type of computations:
  - run Runnable (no argument, no return value)
  - accept Consumer (one argument, no return value)
  - apply Function (one argument, one return value)
- type of compositions (suffix):
  - ∅ one stage
  - either any stage among two
  - both both stages
- exception for thenCombine, which could have been named thenApplyBoth or applyToBoth
Special Method: thenCompose

- Execute a function when a result becomes available (like thenApply).
- With thenApply, this function returns a value that is then implicitly incorporated into a CompletionStage.
- With thenCompose, this function returns a CompletionStage containing the value.
- The distinction is similar to the difference between map and flatMap (in the former, the function returns a value that will be incorporated into a stream, whereas in the latter, it returns a stream of values).
- Useful when using a function (from a third-party library for instance) returning a CompletableFuture.
Example of thenCompose

```java
static class Util {
    static CompletableFuture<Long> add3(int a) {
        return CompletableFuture.supplyAsync(() -> a + 3);
    }
    static CompletableFuture<Long> mult3(int a) {
        return CompletableFuture.supplyAsync(() -> a * 3);
    }
}

Util.add3(10)
    .thenCompose(Util::mult3)
    .thenAccept(System.out::println);
```
Asynchronous Versions

Methods from `CompletionStage` are available in three versions:

1. A default one (no suffix): the following task is executed on the thread executing the previous task or the current thread (e.g. `join`).

2. An asynchronous one (suffix “async”): the following task is submitted to the executor of the current stage.

3. Another asynchronous one with another argument to specify the executor.
Exception Handling

From CompletionStage:

```java
Compl.Stage<T> exceptionally(Function<Throwable,T>)
Compl.Stage<U> handle (BiFunc.<T,Throwable,U>)
Compl.Stage<T> whenComplete (BiConsumer<T,Throwable>)
```

From CompletableFuture:

```java
boolean completeExceptionally(Throwable)
```
Not Only a Java Thing!

```java
id_rpc(&my_server).and_then(|id| {
    get_row(id)
}).map(|row| {
    json::encode(row)
}).and_then(|encoded| {
    write_string(my_socket, encoded)
})
```
Improved Example to Avoid Blocking

```java
CompletableFuture<Integer> c1 = new CompletableFuture<>();
CompletableFuture<Integer> c2 = new CompletableFuture<>();
c1.thenCombine(c2, (y, z) -> System.out.println(y + z));
executor.submit(() -> c1.complete(f(x)));
executor.submit(() -> c2.complete(g(x)));

CompletableFuture<Integer> c1
    = CompletableFuture.supplyAsync(() -> f(x))
CompletableFuture<Integer> c2
    = CompletableFuture.supplyAsync(() -> g(x))
c1.thenAcceptBoth(c2, (y, z) -> System.out.println(y + z));
```
Outline

Blocking Problem

Future Composition

Summary and References
Future Composition

- Making methods asynchronous (returning before all their work is done) allows additional parallelism and reduce blocking.
- The `CompletableFuture` class expresses one-shot asynchronous computations. Combinators can be used to compose asynchronous computations to reduce blocking.
- A `CompletableFuture` allows propagating and managing errors generated within an asynchronous task.
- A synchronous API can be asynchronously consumed by wrapping invocations into a `CompletableFuture`.

Summary and References
Official Documentation

- Documentation of class CompletetableFuture
- Documentation of interface CompletionStage