Programming in Java

3. Multi-Threading 2

Java Multi-Threading

- As Java has evolved, there have been several versions of how Java implements multi-threading
  - The multi-threading show in module 1 is the older version of multi-threading
  - Seen a lot of Java legacy code
- The more modern approach
  - Gets rid of a lot of the boiler plate code by not requiring us to write code to manage threads directly
  - Instead, the overhead of creating and running threads is moved to the Java class libraries
  - Instead we use threads through various interfaces
  - This also is more efficient from a resource utilization perspective
  - Java can optimize the number of threads running and ensure thread safety

The Resource Problem

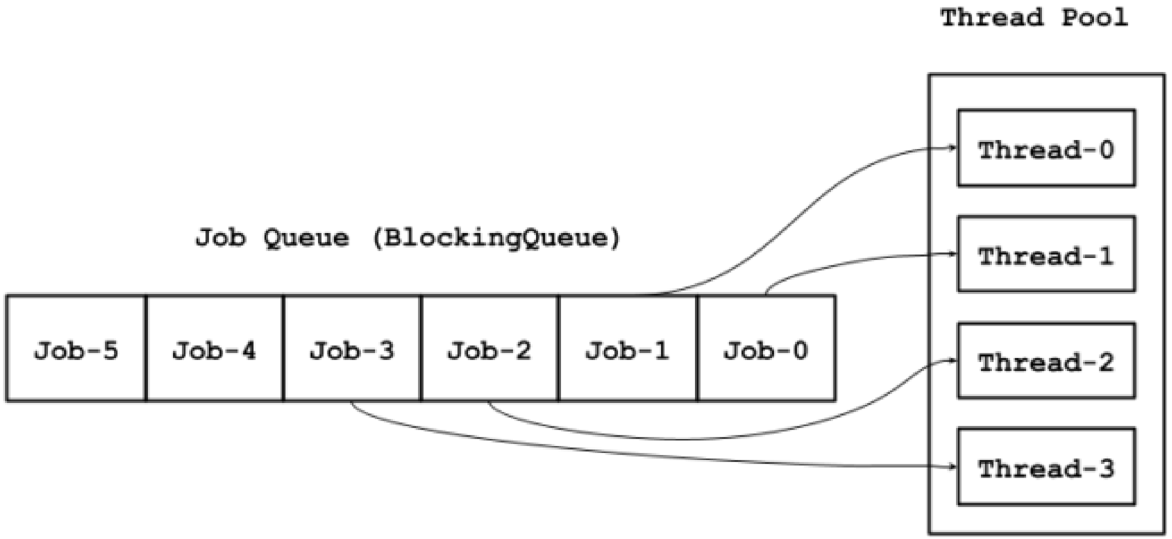
- Threads are often used for task that are:
  - Sort in duration
  - Called very frequently
- The problems with managing any kind of resource with these characteristics are
  - The amount of time spent creating and shutting down threads starts to become significant – the system starts to “thrash” trying to manage the threads
  - Too many threads can cause out of memory issues
- Managing threads at the low level we have been doing:
  - Requires writing a lot of boilerplate code
  - Is time consuming and error prone
- The solution is to delegate the actual creating and running of threads to the JRE

Pooling Resources

- Resource pool - collection of pre-created resources that are available on demand
  - This is a standard architectural strategy
  - Flyweight design pattern
- A thread pool contains a number of pre-created threads
  - Delay introduced by thread creating is eliminated, a thread is just selected from the pool
  - The thread is passed a Runnable object and then executes it
  - When the thread finishes executing, it is returned to the pool to be reused later
- Reduces thread life-cycle overhead and thrashing
- Allows resource limits to be set, like the maximum number of threads
- Allows programmers to concentrate on the executable code inside the run() method instead of the overhead of thread management
- This is the preferred way to create and use threads in modern Java programming

Executor Services

- The Executor interface
  - Creates a pool of threads and a queue to hold jobs
  - A runnable object to be executed in a thread is called a “job”
  - When a request comes in via a job queue, it is allocated to a thread which performs the task
  - When the task is finished, the thread is returned to the pool



Executor Services

- Java provides a concurrency library which supports a built-in Java thread pool
- Implemented as three interfaces
- An *Executor* interface that provides a replacement to the standard thread syntax.
  - *(new Thread(runnablecode).start()* can be replaced by *e.execute(runnablecode)* where e is an instance of Executor
- *ExecutorService* interface extends the *Executor* interface to include a submit method for a run() method which returns a value
  - We will not be covering this interface in this class
- *ScheduledExecutorService* which adds methods to allow scheduling of threads

Executor Services

- Start by allocating a Thread pool using one of the constructors (factory method)
  - The following code implements a fixed size Executor service
  - The shutdown() message
    - *Stops the service from accepting new tasks*
    - *Shuts the service down when all the executing threads have exited*

```
    public static void main(String[] args) {

        // Creates a new Thread Pool with 3 executors
        ExecutorService myPool = Executors.newFixedThreadPool(3);

        // Shuts the pool down once all the threads have terminated
        myPool.shutdown();

    }
}
```

Submit a Task

- Once the pool is started
  - Runnable tasks are submitted via the execute() method
  - Execute queues up the task, and when a thread is available, passes the task to the thread then executes the start() method on the thread

```java
// Creates a new Thread Pool with 3 executors

ExecutorService myPool = Executors.newFixedThreadPool(3);
for (int i = 1; i < 5; i++) {
    myPool.execute(new MyTask("Task " + i));
}
// Shuts the pool down once all the threads have terminated
myPool.shutdown();
```

Customized Executor

- We can also create our own service with customized parameters
  - Core threads – the number of threads to start with
  - Max threads – the number of threads that can the executor service can scale up to
  - Keep alive – the amount of time to keep an executor running when idle
  - Time units – the time units used to measure the keep alive
  - BlockingQueue – the queue object to be used by the pool

- Executors are designed to be highly configurable

Customized Executor Submission

- Exactly the same as before
- The parameters are tuned for performance based on our requirements
- And based on performance history

```
public static void main(String[] args) {

    int corePoolSize = 3;
    int maxPoolSize = 5;
    long keepAliveTime = 3000;
    BlockingQueue<Runnable> pool = new ArrayBlockingQueue<Runnable>(100);


    ExecutorService myPool = new ThreadPoolExecutor(
            corePoolSize, maxPoolSize, keepAliveTime, TimeUnit.MILLISECONDS,pool);
```

Lab 3-1

Executors

Futures

- Java programs often run tasks in separate threads.
  - In the examples we have seen so far, runnable tasks have been of the form `void run()`
  - But if we want to provide arguments and get a return value
- Basic issues to be solved are
  - How to start a task with parameters
  - How to determine when the task is finished
  - How we can get back the return value when the task is finished
- Futures are Java’s way to handle asynchronous results.
  - Executors are designed to be highly configurable in order to handle Futures

Futures

- A `Future<T>` represents the result of an asynchronous computation.
  - Think of it like a placeholder for a value that’s coming later.
  - The `T` is the type of data returned by the task
- For example
  - Future<Integer> result = executor.submit(() -> 42);
  - Submits a Lambda (runnable) to an executor that returns the value 42
  - When the task finishes, the value is stored in the result variable
  - You can think of the future as a box that will, at some point, hold a value
- You can:
  - Check if it's done → future.isDone()
  - Block and get the result → future.get()

Futures

- A `Future<T>` represents the result of an asynchronous computation.
  - Think of it like a placeholder for a value that’s coming later.
  - The `T` is the type of data returned by the task
- For example
  - Future<Integer> result = executor.submit(() -> 42);
  - Submits a Lambda (runnable) to an executor that returns the value 42
  - When the task finishes, the value is stored in the result variable
  - You can think of the future as a box that will, at some point, hold a value
- You can:
  - Check if it's done → future.isDone()
  - Block and get the result → future.get()

Lab 3-2

Futures

Completable Futures

- Futures are limited
  - They do not handle exceptions
  - They do not “chain” meaning one task can call another task
  - Excellent for small individual tasks to be run in a thread
- CompletableFuture extends the capabilities of Future by allowing:
  - Non-blocking operations
  - Chaining of dependent tasks
  - Combining multiple futures
  - Handling results and exceptions fluently
  - Manual completion and cancellation
- Heavily influenced by reactive streams programming
  - Not covered in this course

Completable Futures

- Uses the ForkJoinPool
  - ForkJoinPool is a specialized thread pool designed for parallelism
  - Optimized for "divide-and-conquer" tasks, where large tasks are split (forked) into subtasks and then combined (joined).
- How It Works
  - Tasks are submitted to the pool.
  - Each thread in the pool has its own deque (double-ended queue).
  - When a thread finishes its work, it tries to steal a tasks waiting in other threads deques
  - Unlike an executor, each thread has it’s own task buffer so there is no thread contention for a shared buffer

Completable Futures

- Downsides
  - Be careful overusing the common pool
  - It’s shared, so heavy tasks may delay unrelated async operations.
- Avoid blocking operations in tasks submitted to the common pool.
  - It’s optimized for CPU-bound tasks, not I/O.
  - Consider a custom thread pool for I/O-heavy or long-running tasks.

Non-Blocking Architecture

- Future.get() blocks the calling thread until the result is ready
  - CompletableFuture allows reacting to results asynchronously using callback methods
  - These are methods like .thenApply(), .thenAccept()
  - These take the result and then respond to it
  - Improves thread efficiency
  - No threads are sitting idle waiting for results.
- The callbacks do not block the code
  - The code continues to execute
  - At some point the task finished, then the callback is executed

Non-Blocking Example

- *CompletableFuture.supplyAsync(() -> fetchFromDB())*
  - Submits *fetchFromDB()* to run asynchronously in the default ForkJoinPool.commonPool()
  - Returns immediately with a *CompletableFuture<T>* which is empty
  - The main thread continues execution without waiting.
  - *.thenAccept(result -> updatePage(result))* is a callback to be triggered when the async result is ready.
  - The code *updatePage(result)* will not run immediately, but will run after *fetchFromDB()* completes

```
CompletableFuture.supplyAsync(() -> fetchFromDB())
    .thenAccept(result -> updatePage(result));
```

Chaining and Task Pipelines

- A task pipeline
  - Each task processes the result of the previous one.
- CompletableFuture enables this fluently using methods like:
  - thenApply() – transforms result
  - thenAccept() – consumes result
  - thenRun() – runs next action regardless of result

```
CompletableFuture.supplyAsync(() -> getUserId())
    .thenCompose(id -> fetchProfile(id))
    .thenApply(profile -> enrich(profile))
    .thenAccept(this::sendToClient);
```

Composability

- Combine multiple independent or dependent tasks using:
  - *thenCombine()* – combines results of two independent tasks
  - *thenCompose()* – chains dependent async tasks
  - *allOf() / anyOf()* – coordinates many task
- Enables parallelism:
  - run tasks concurrently when they don't depend on each other.
- Enables complex workflows:
  - define multi-stage pipelines involving multiple results.

```
CompletableFuture<Integer> price = getPriceAsync();
CompletableFuture<Integer> tax = getTaxAsync();

price.thenCombine(tax, (p, t) -> p + t)
     .thenAccept(System.out::println);
```

Error Handling

- Can catch and respond to exceptions using:
  - exceptionally() – supply fallback
  - handle() – process result or exception
  - whenComplete() – log or audit outcomes
- Avoids unchecked exceptions crashing the app
- Allows graceful recovery (fallbacks, retries)
- Centralizes error flow logic

```
fetchData()
    .thenApply(data -> parse(data))
    .exceptionally(ex -> {
        logError(ex);
        return defaultData();
    });
```

Asynch

- Synchronous Methods
  - thenApply, thenAccept, etc.
  - Run the next step in the same thread that completed the previous stage.
- Asynchronous Methods
  - thenApplyAsync(), thenAcceptAsync()
  - These methods always run the next step in a different thread
- Notice in the examples
  - The tasks are all provided as Lambda functions

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    System.out.println("Task 1 in: " + Thread.currentThread().getName());
    return "Hello";
}).thenApply(result -> {
    System.out.println("Task 2 in: " + Thread.currentThread().getName());
    return result + " World";
});
```

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    System.out.println("Task 1 in: " + Thread.currentThread().getName());
    return "Hello";
}).thenApplyAsync(result -> {
    System.out.println("Task 2 in: " + Thread.currentThread().getName());
    return result + " World";
});
```

Lab 3-3

Completable Futures

Java™