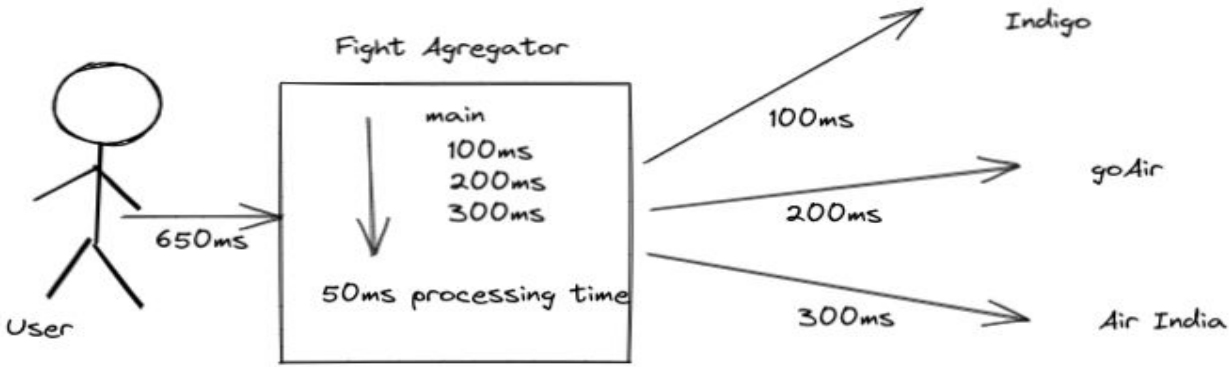
Java Backend Development Live-85

lecture-6

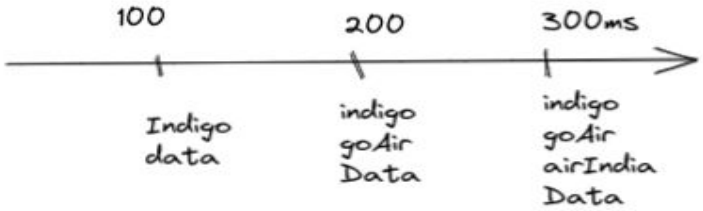
Agenda

- Executors Framework
- ExecutorService & ThreadPoolExecutor
- Concurrency Issues
- Synchronized and volatile
- Callable and Future
- Wait and Notify

Example: flight aggregator



Using 3 thread.
300ms All data will available.
API response time:350ms



Executors Framework (Thread Pool)

A framework for creating and managing threads.

- **Thread Creation**: It provides various methods for creating threads and pool of threads.
- **Thread Management**: It manages the life cycle of the threads in the thread pool.
- **Task submission and execution**: Executors framework provides methods for submitting tasks for execution in the thread pool.

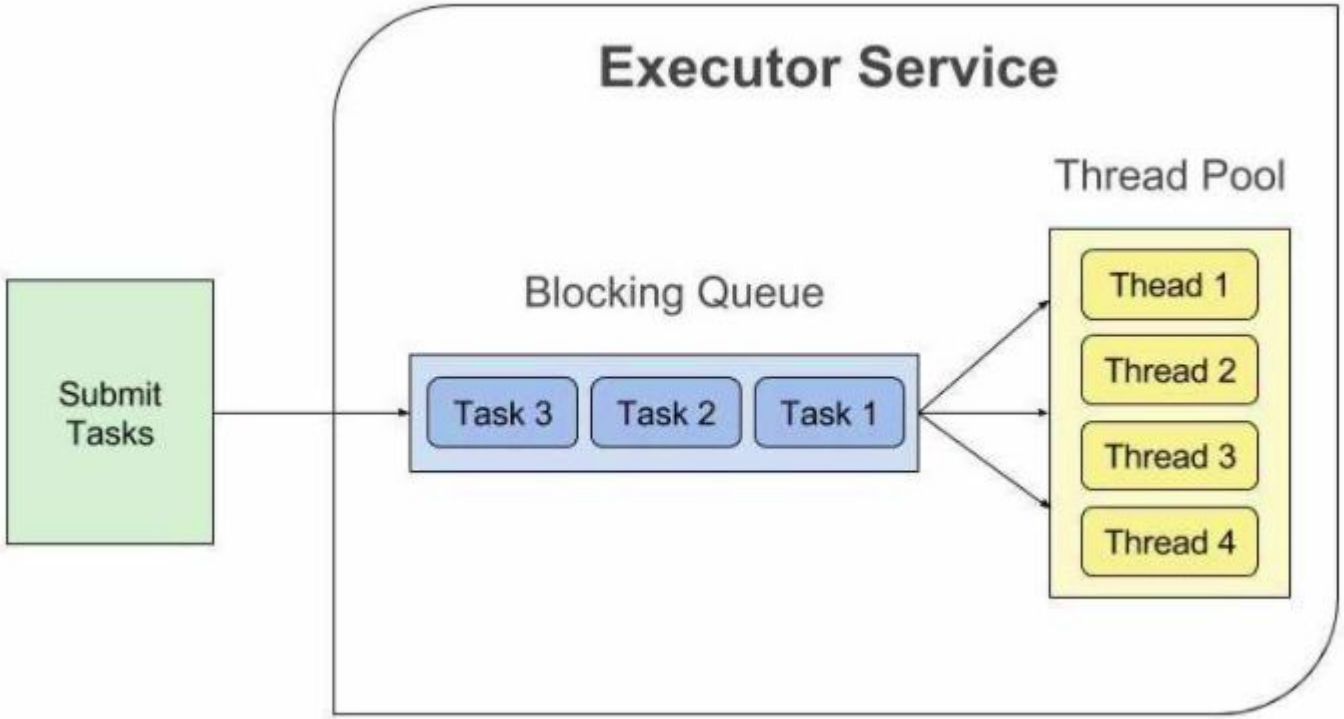
ExecutorService

```
ExecutorService fixExecutorService = Executors.newFixedThreadPool(5);
ExecutorService executorService = Executors.newSingleThreadExecutor();
        executorService.submit(() -> {
            System.out.println("Task Running in : " + Thread.currentThread().getName());
        });
        executorService.shutdown();
```

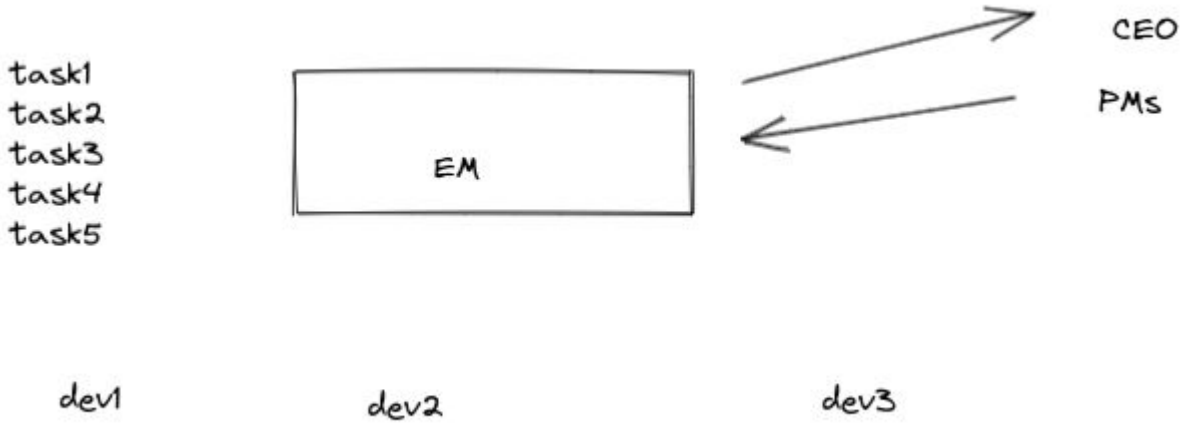
- shutdown() - when `shutdown()` method is called on an executor service, it stops accepting new tasks, waits for previously submitted tasks to execute, and then terminates the executor.
- shutdownNow() - this method interrupts the running task and shuts down the executor immediately.

Thread Pool

- Creating a thread is an expensive operation and it should be minimized.
- Having worker threads minimizes the overhead due to thread creation because executor service has to create the thread pool only once and then it can reuse the threads for executing any task.
- Tasks are submitted to a thread pool via an internal queue called the ***Blocking Queue***.


Executor Service
Thread Pool
Blocking Queue
Submit Tasks
Task 3
Task 2
Task 1
Thead 1
Thread 2
Thread 3
Thread 4

Engineering Manager as ExecutorService and developers as worker threads.



ThreadPoolExecutor

```
int  corePoolSize  =   5;
int  maxPoolSize   =   10;
long keepAliveTime = 5000;
ExecutorService threadPoolExecutor =
        new ThreadPoolExecutor(
                corePoolSize,
                maxPoolSize,
                keepAliveTime,
                TimeUnit.MILLISECONDS,
                new LinkedBlockingQueue<Runnable>()
        );
```

Concurrency Issues

Two types of problems arise when multiple threads try to read and write shared data concurrently -

- Thread interference errors (Race Conditions)
- Memory consistency errors

Thread interference errors (Race Conditions)

When multiple threads try to read and write a shared variable concurrently, and these read and write operations overlap in execution, then the final outcome depends on the order in which the reads and writes take place, which is unpredictable. This phenomenon is called **Race Condition**.

E.g. Multiple thread incrementing total visitor count.

Memory consistency errors

Memory inconsistency errors occur when different threads have inconsistent views of the same data. This happens when one thread updates some shared data, but this update is not propagated to other threads, and they end up using the old data

Processors also try to optimize things, for instance, a processor might read the current value of a variable from a **temporary register**, instead of **main memory**.

E.g. One Thread is signaling another thread to stop by changing value of a variable.

Synchronization

Thread interference and memory consistency errors can be avoided by ensuring the following two things-

- Only one thread can read and write a shared variable at a time. When one thread is accessing a shared variable, other threads should wait until the first thread is done. This guarantees that the access to a shared variable is ***Atomic***, and multiple threads do not interfere.
- Whenever any thread modifies a shared variable, it automatically establishes a ***happens-before*** relationship with subsequent reads and writes of the shared variable by other threads. This guarantees that changes done by one thread are visible to others.

Volatile Keyword

- Volatile keyword is used to avoid memory consistency errors in multithreaded programs.
- Volatile variable’s value will always be read from the main memory instead of temporary registers.
- It does not guarantee atomicity for compound operations.

Callable and Future

- A `Callable` is similar to `Runnable` except that it can return a result and throw a checked exception.
- The concept of `Future` is similar to Promise in other languages like Javascript. It represents the result of a computation that will be completed at a later point of time in future.
- `ExecutorService.submit()` method returns immediately and gives you a Future.
- The `get()` method blocks until the task is completed. The `Future` API also provides an `isDone()` method to check whether the task is completed
- The `future.get()` method will throw a `TimeoutException` if the task is not completed within the specified time.

Wait and Notify

- `wait()` causes the current thread to wait **until another thread invokes `notify()` or `notifyAll()`** on the same object.
- `notify()` wakes up a single thread that's waiting on that object's monitor.
- Both `wait()` and `notify()` must be called **from a synchronized block or method**.