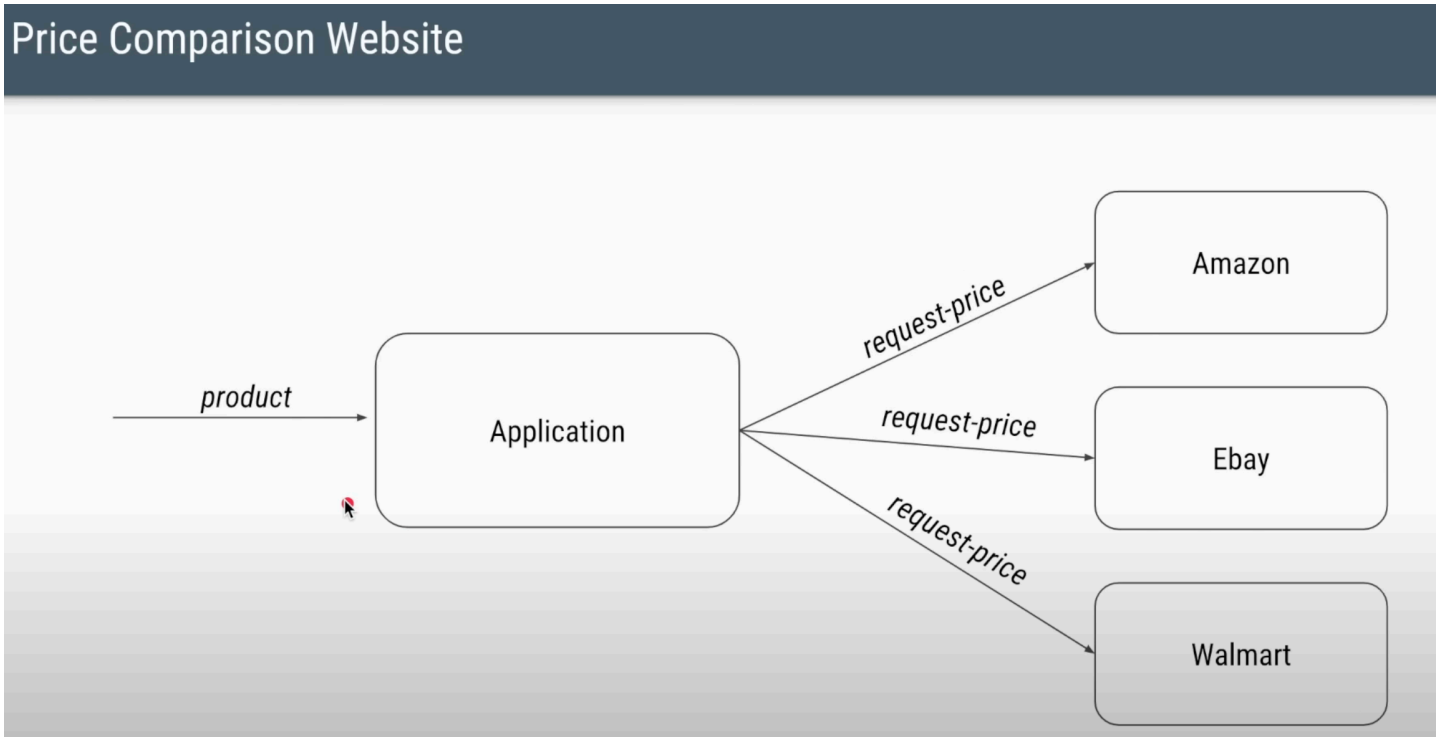
Multithreading : Interview Questions Part 2

Question1:

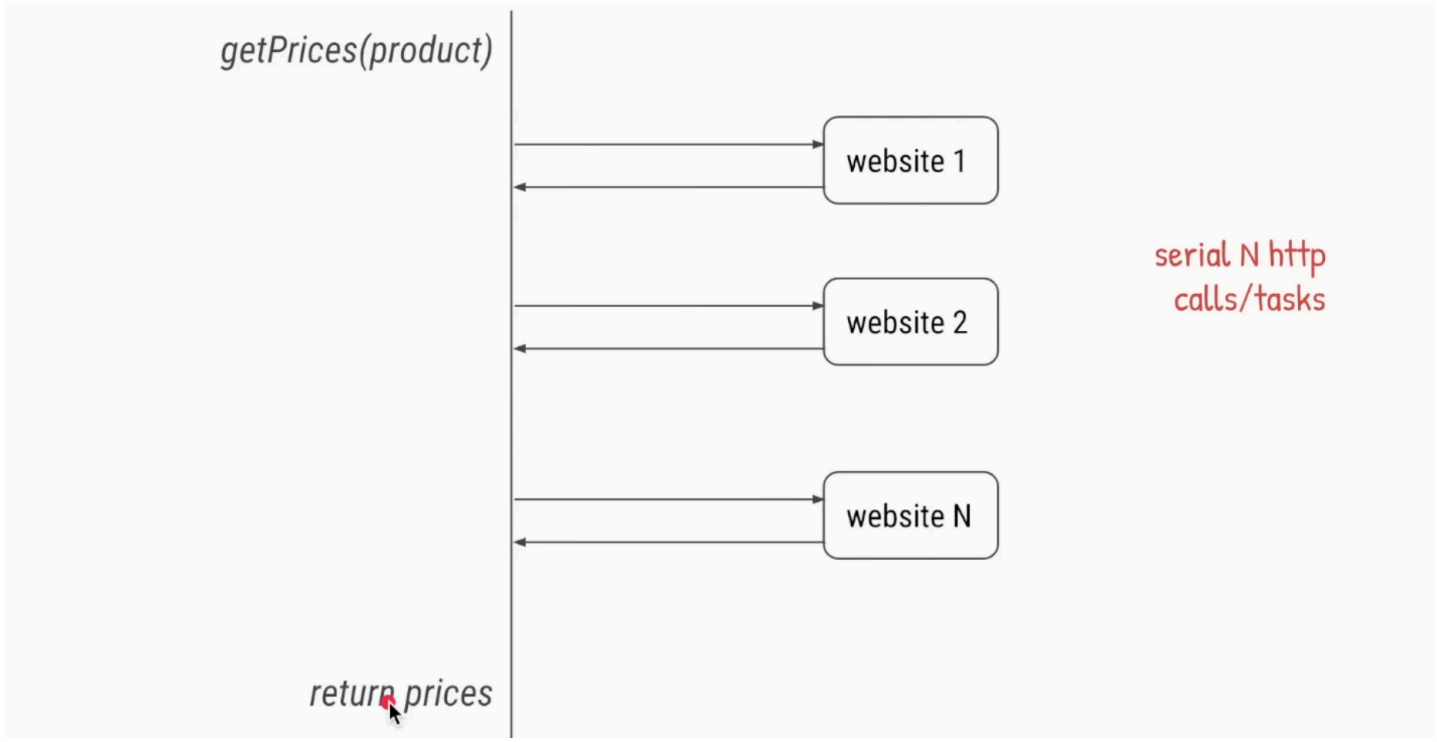
Implement Scatter Gather pattern



Press "/" to insert an element

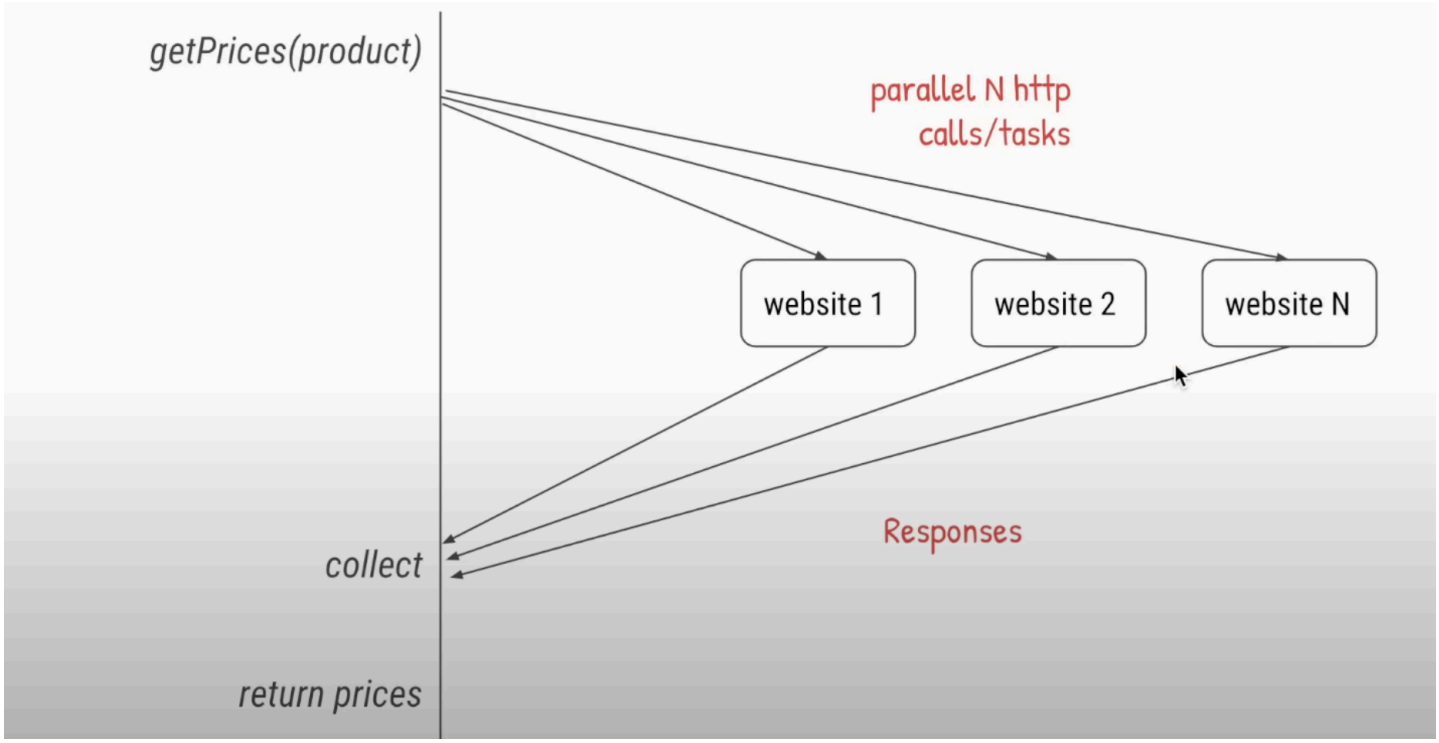
write code to retrieve prices from N external resources:

Not efficeint solution:

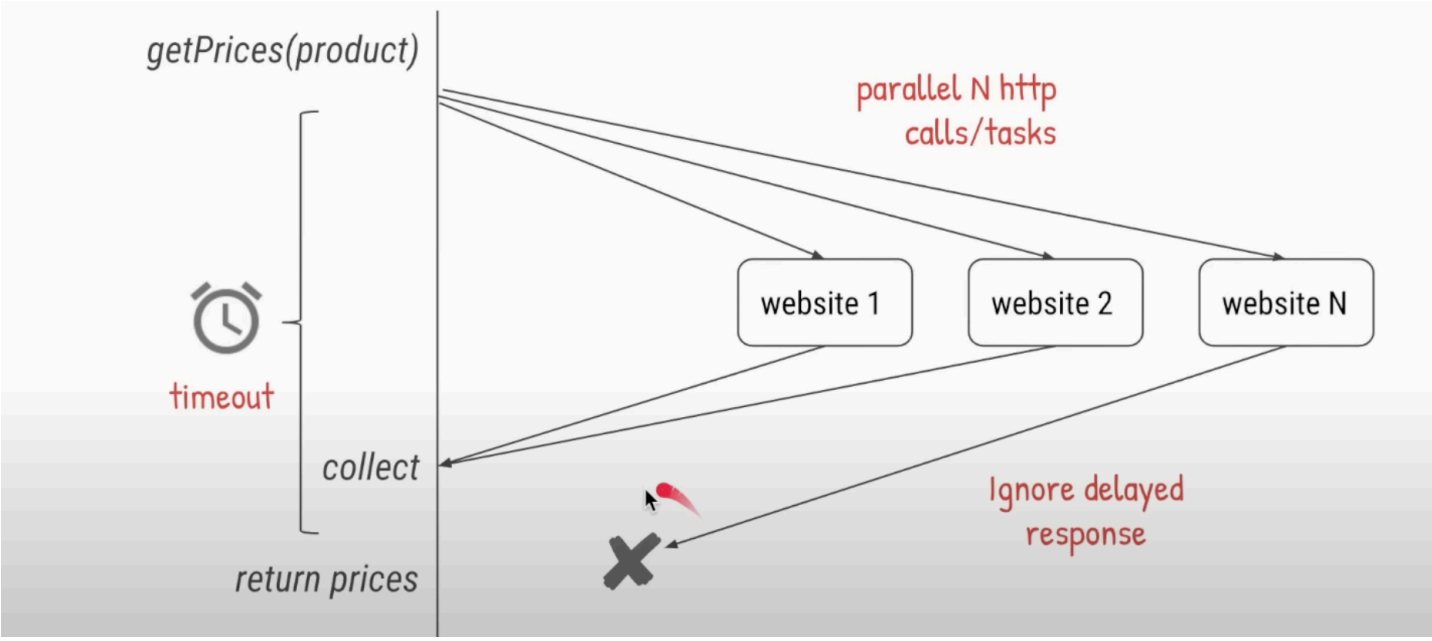


efficient solution :

Do || calls



write code to retrieve prices from N external resources given that 3 sec timeout:



Break down:

- Trigger and wait for N tasks
- Add timeout

Simple approach:

```
ExecutorService threadPool = Executors.newFixedThreadPool(4);

private Set<Integer> getPrices(int productId) {

    threadPool.submit(new Task(url1, productId));
    threadPool.submit(new Task(url2, productId));
    threadPool.submit(new Task(url3, productId));
}

private class Task implements Runnable {

    private String url;
    private int productId;

    public Task(String url, int productId) {
        this.url = url;
        this.productId = productId;
    }

    @Override
    public void run() {
        // make http call to get price
    }
```

raw threads can also be used instead

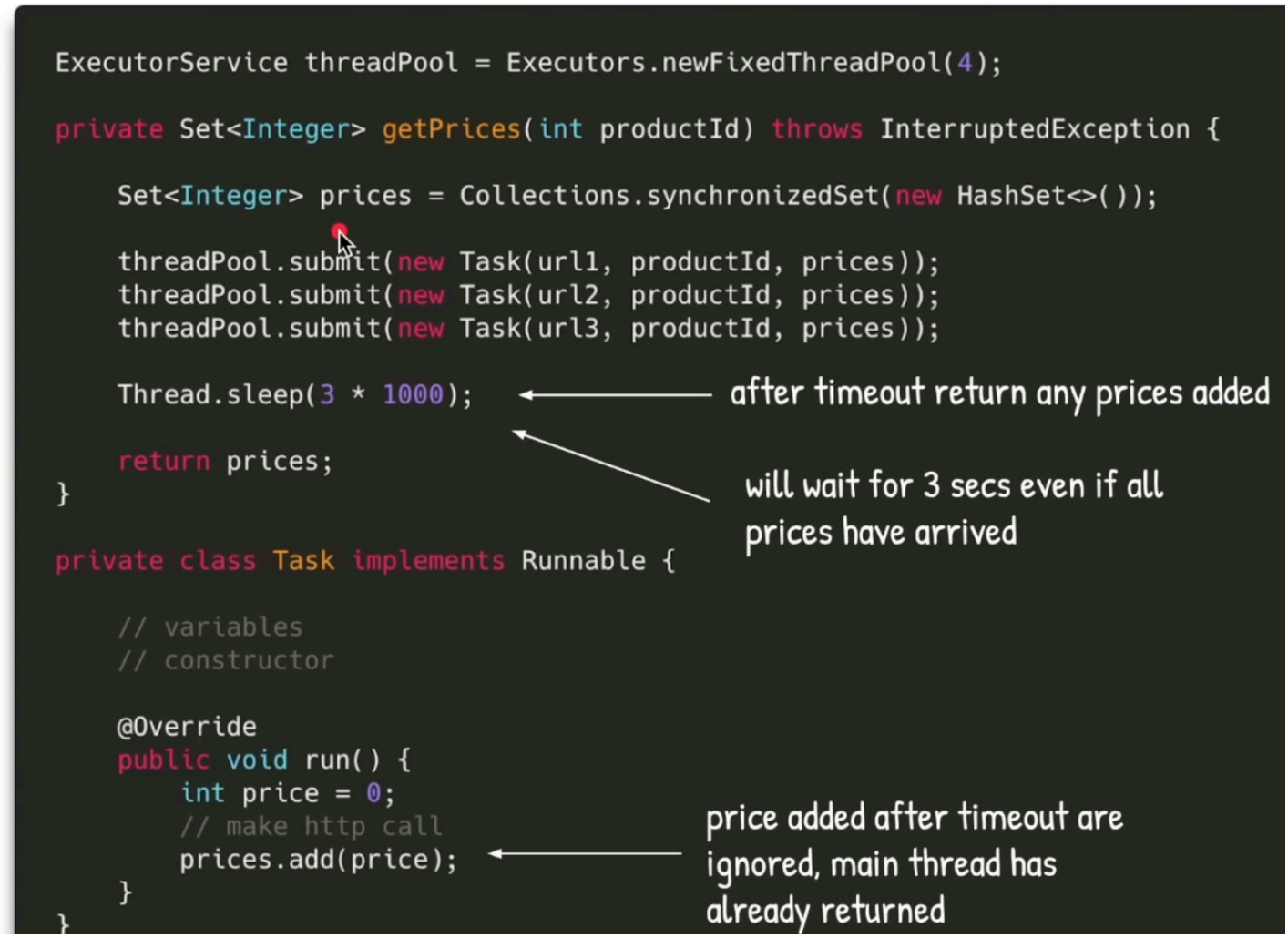
Add prices to common collection

```
ExecutorService threadPool = Executors.newFixedThreadPool(4);

private Set<Integer> getPrices(int productId) {

    Set<Integer> prices = Collections.synchronizedSet(new HashSet<>());

    threadPool.submit(new Task(url1, productId, prices));
    threadPool.submit(new Task(url2, productId, prices));
    threadPool.submit(new Task(url3, productId, prices));

    return prices;
}

private class Task implements Runnable {

    private String url;
    private int productId;
    private Set<Integer> prices;

    public Task(String url, int productId, Set<Integer> prices) {
        // ...
        this.prices = prices;
    }

    @Override
    public void run() {
        int price = 0;
        // make http call
        prices.add(price);
    }
```

synchronized for thread-safety

add price to common collection

Waiting for 3 seconds (for every single request)



Option 2: countdown latch

Give tasks latch to countdown

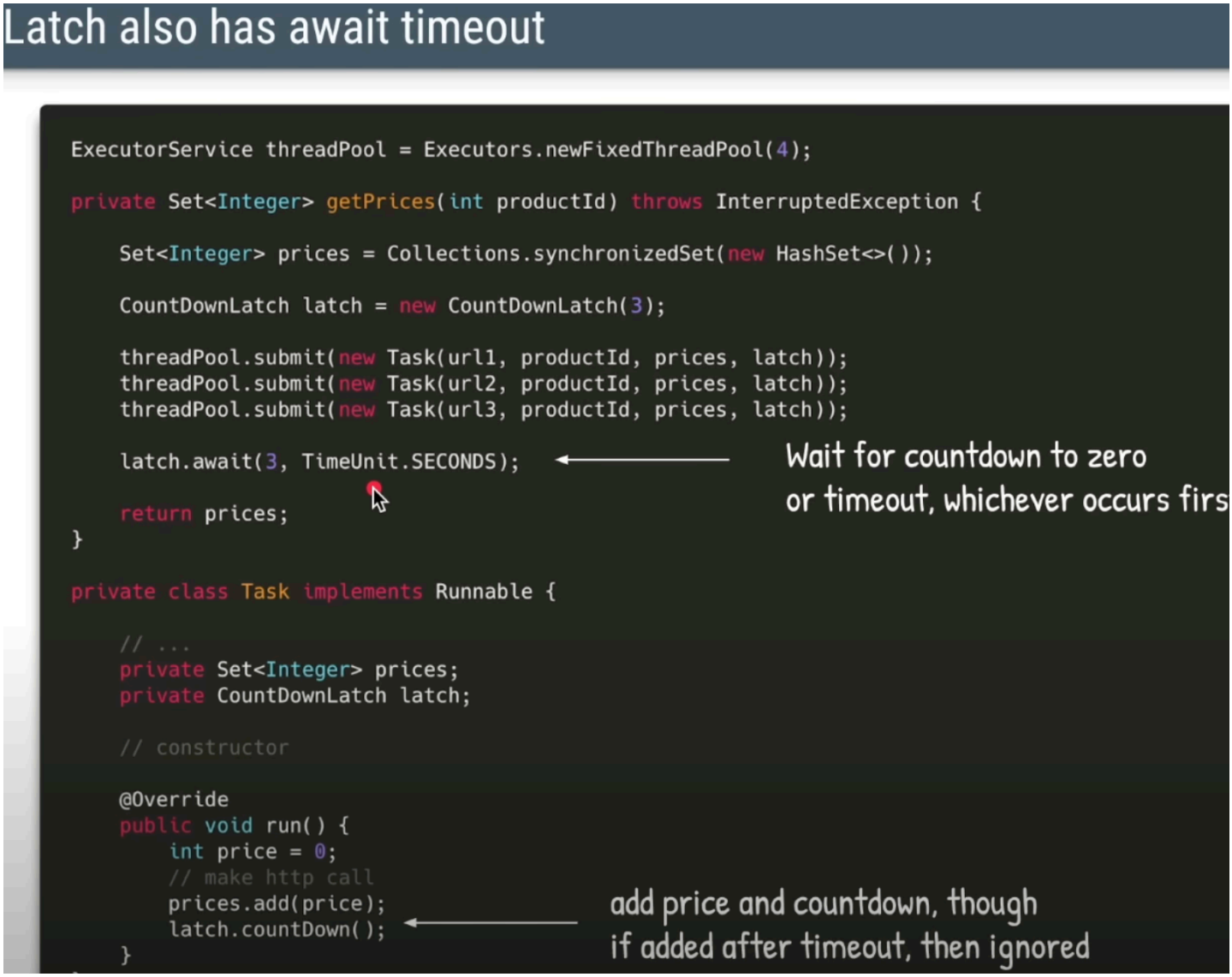
```
ExecutorService threadPool = Executors.newFixedThreadPool(4);

private Set<Integer> getPrices(int productId) throws InterruptedException {

    Set<Integer> prices = Collections.synchronizedSet(new HashSet<>());

    CountDownLatch latch = new CountDownLatch(3);

    threadPool.submit(new Task(url1, productId, prices, latch));
    threadPool.submit(new Task(url2, productId, prices, latch));
    threadPool.submit(new Task(url3, productId, prices, latch));

    latch.await();

    return prices;
}

private class Task implements Runnable {

    private String url;
    private int productId;
    private Set<Integer> prices;
    private CountDownLatch latch;

    // constructor

    @Override
    public void run() {
        int price = 0;
        // make http call
        prices.add(price);
        latch.countDown();
```

initialized to N, for N task

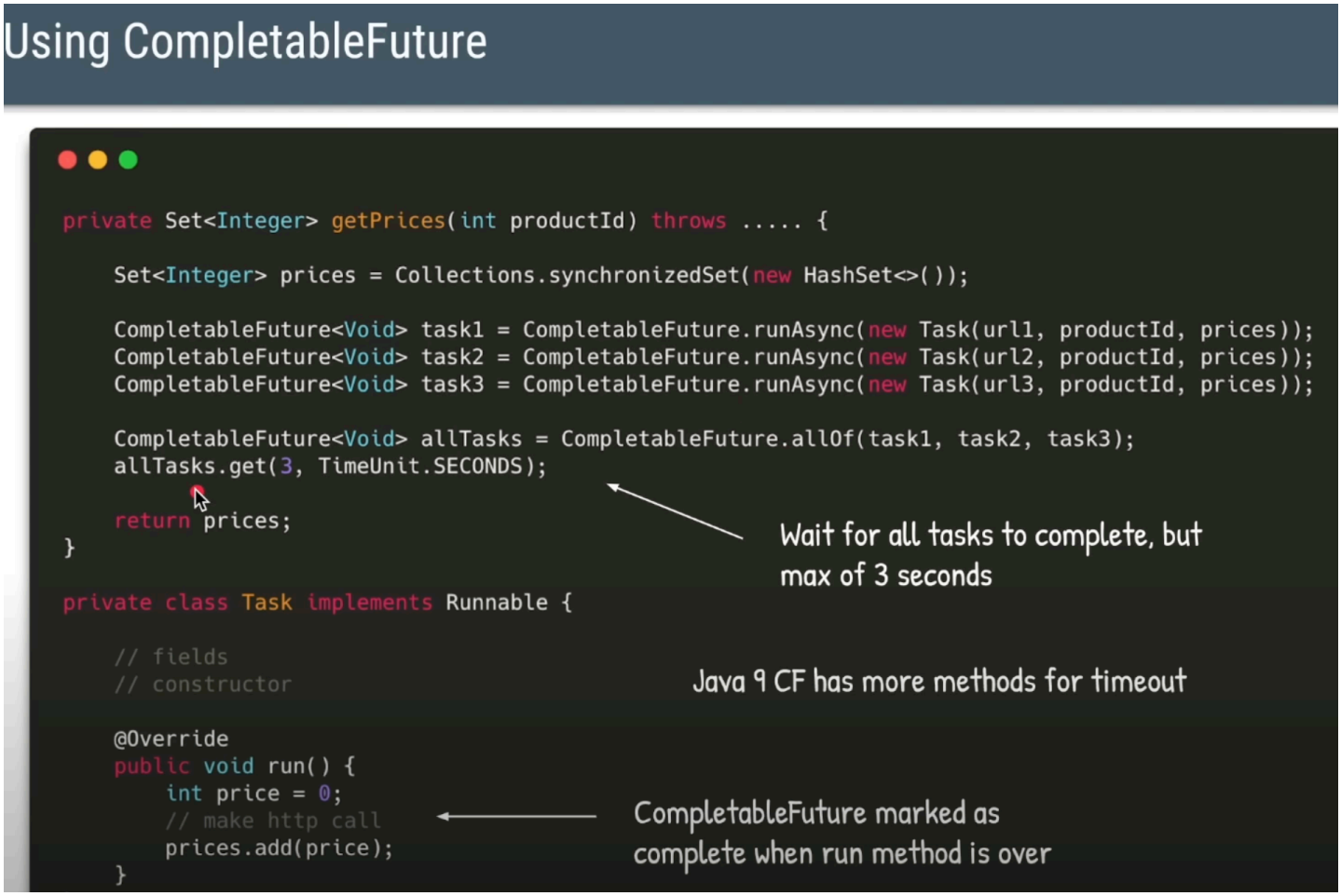
give latch to each tas

Wait for every task to countdown
(ie wait for count = 0)

add price and countdown



Option 3 : Completable Future:



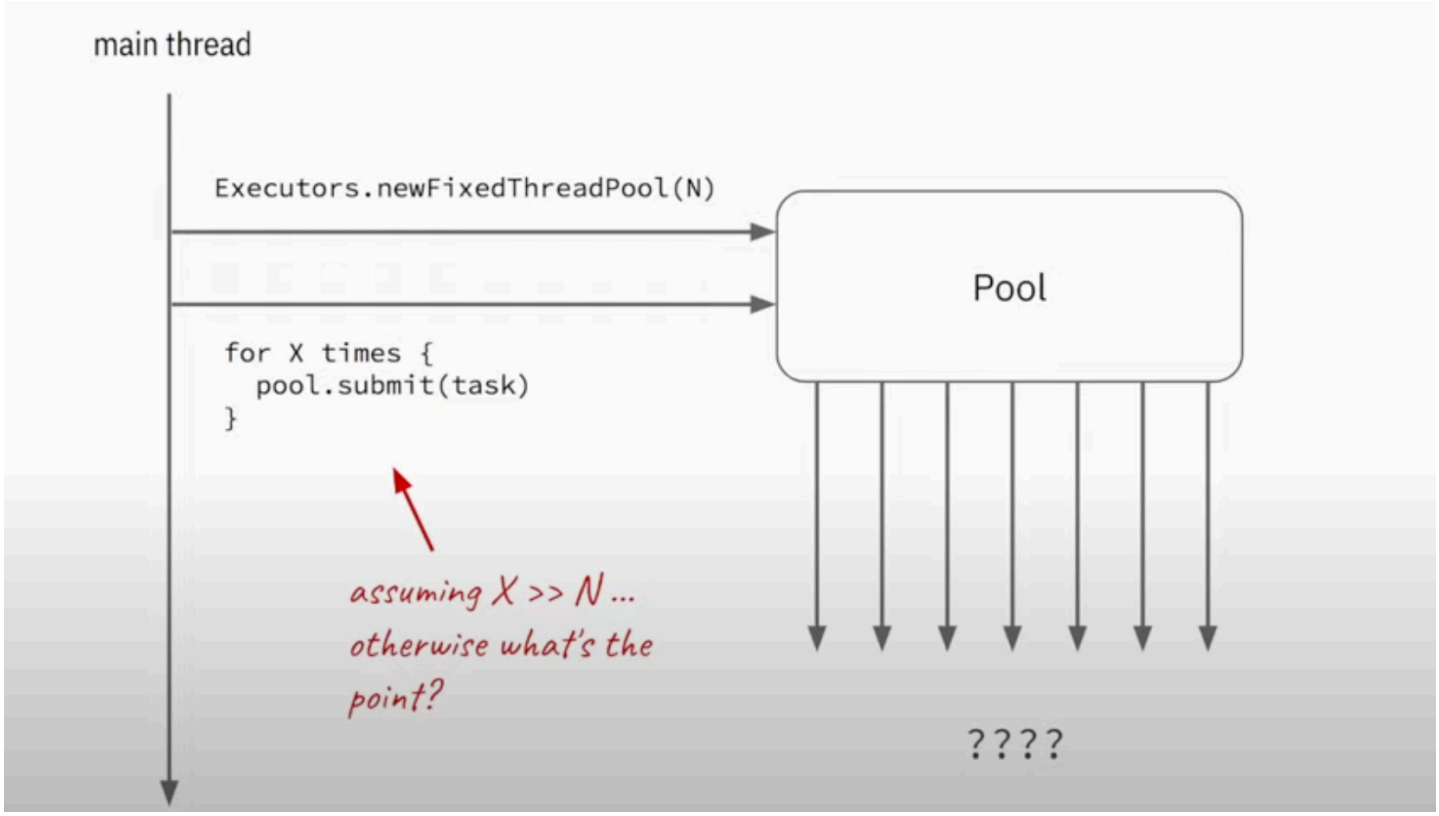
Other solution:

- Phaser
- Condition and Locks

Question : What is ideal thread pool size

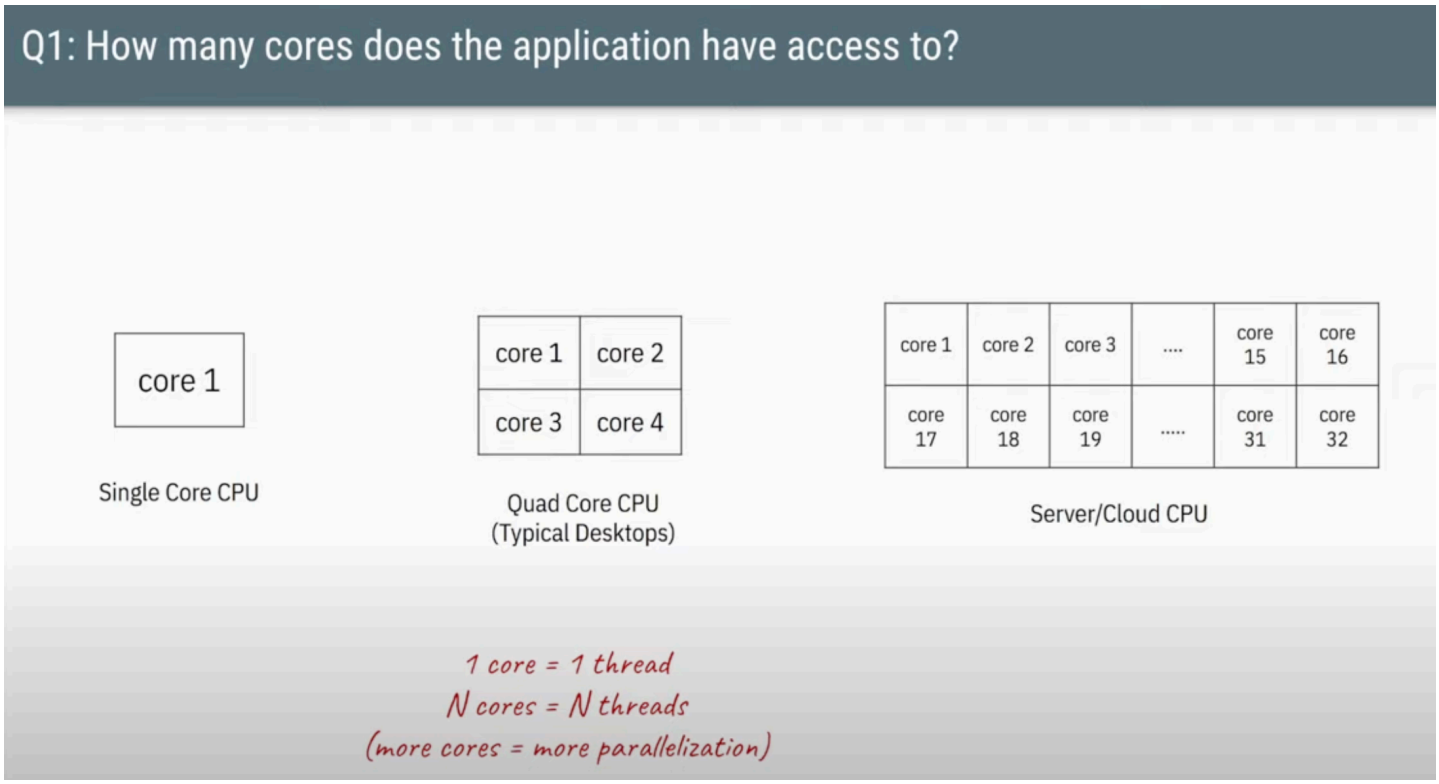
Assumptions:

- apply for only fixed threadpool bcoz in the cached thread pool we can create as many thread we want
- More task to handle rather than only executing 1-2 tasks



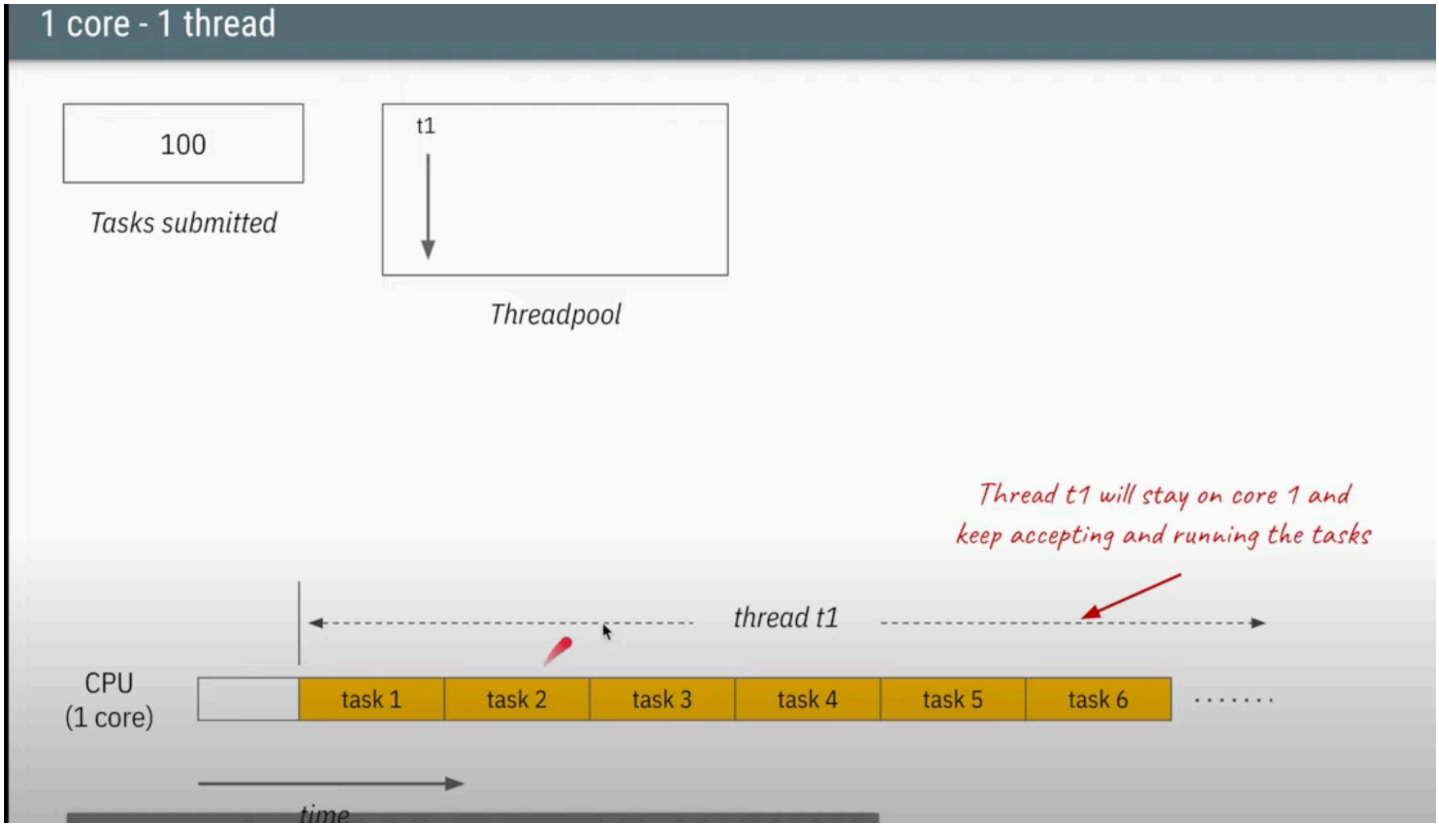
- **So there is no ideal number of thread pool , All depends on the use cases.**

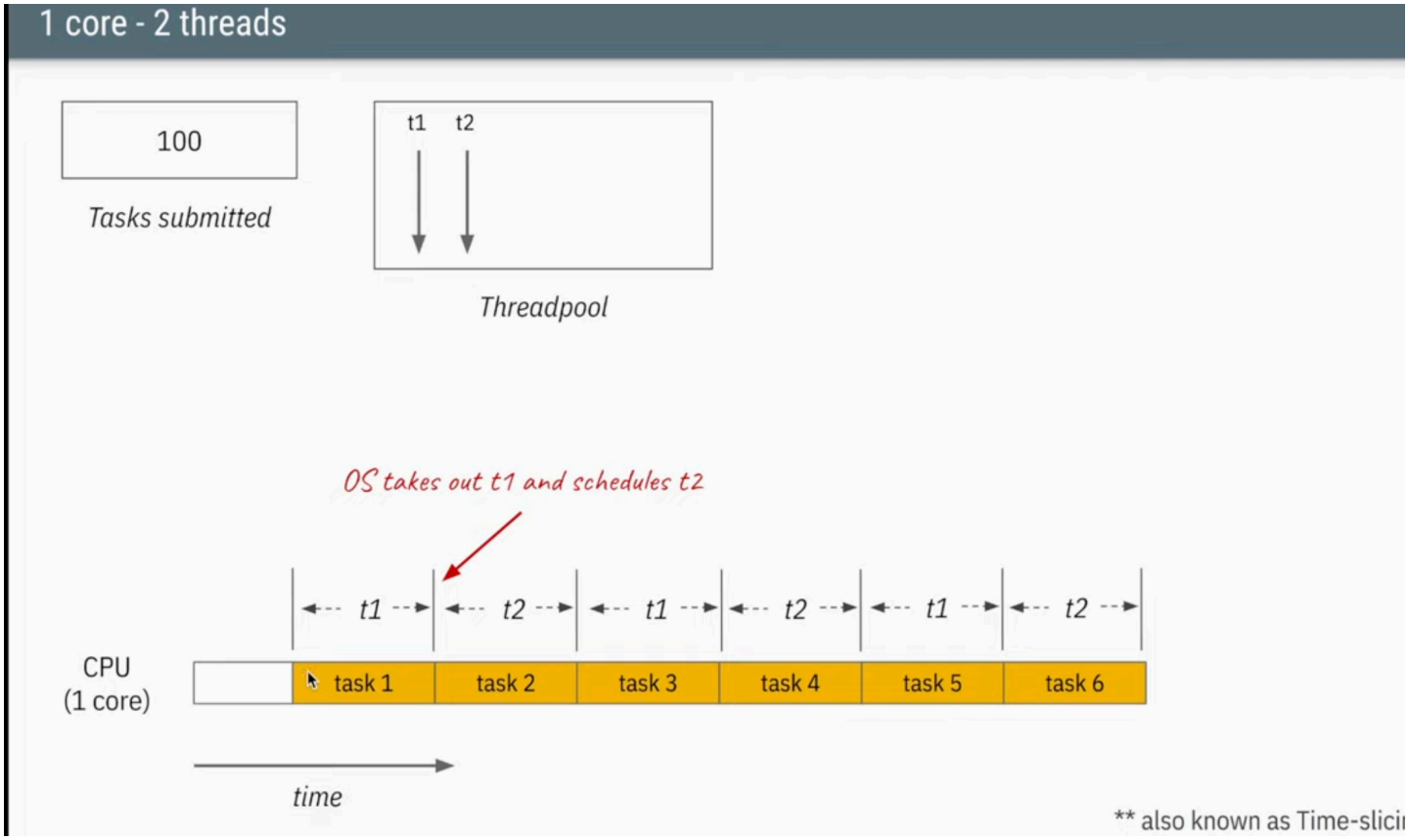
Factor involved in deciding this :



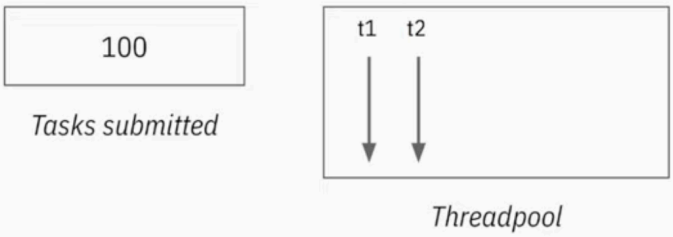
2. **Types of tasks**

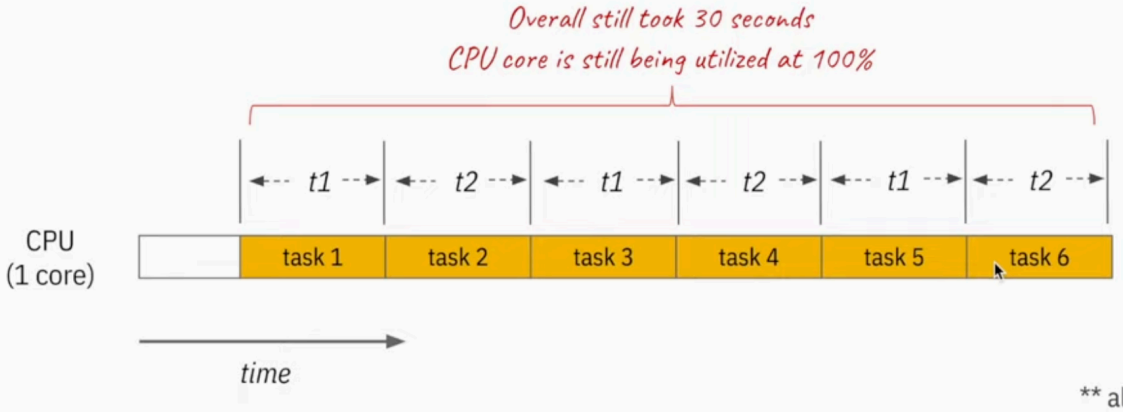
CPU intensive task





1 core - 2 threads



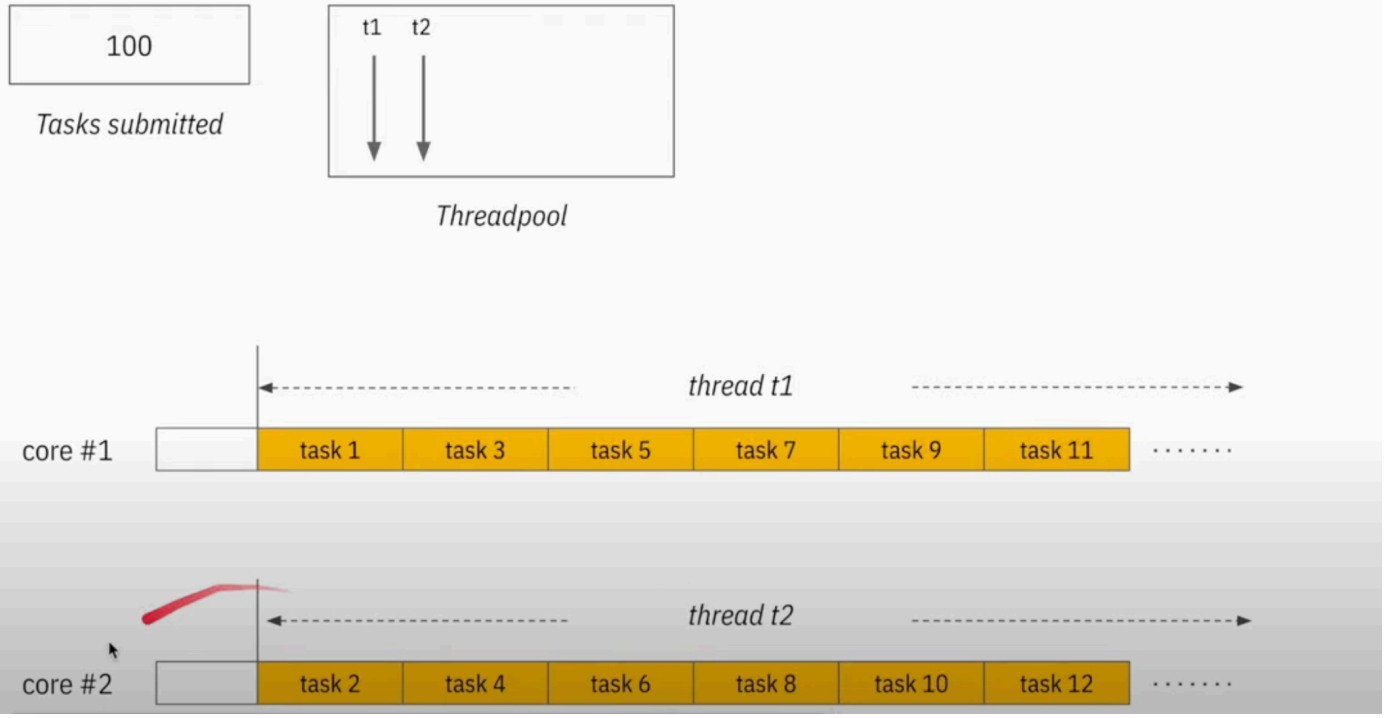


** also known as Time-slici
** this scheduling can happen even in middle of task executi

CPU bound tasks

- With same number of cores, increasing thread count doesn't speed up task completion.

2 core - 2 threads

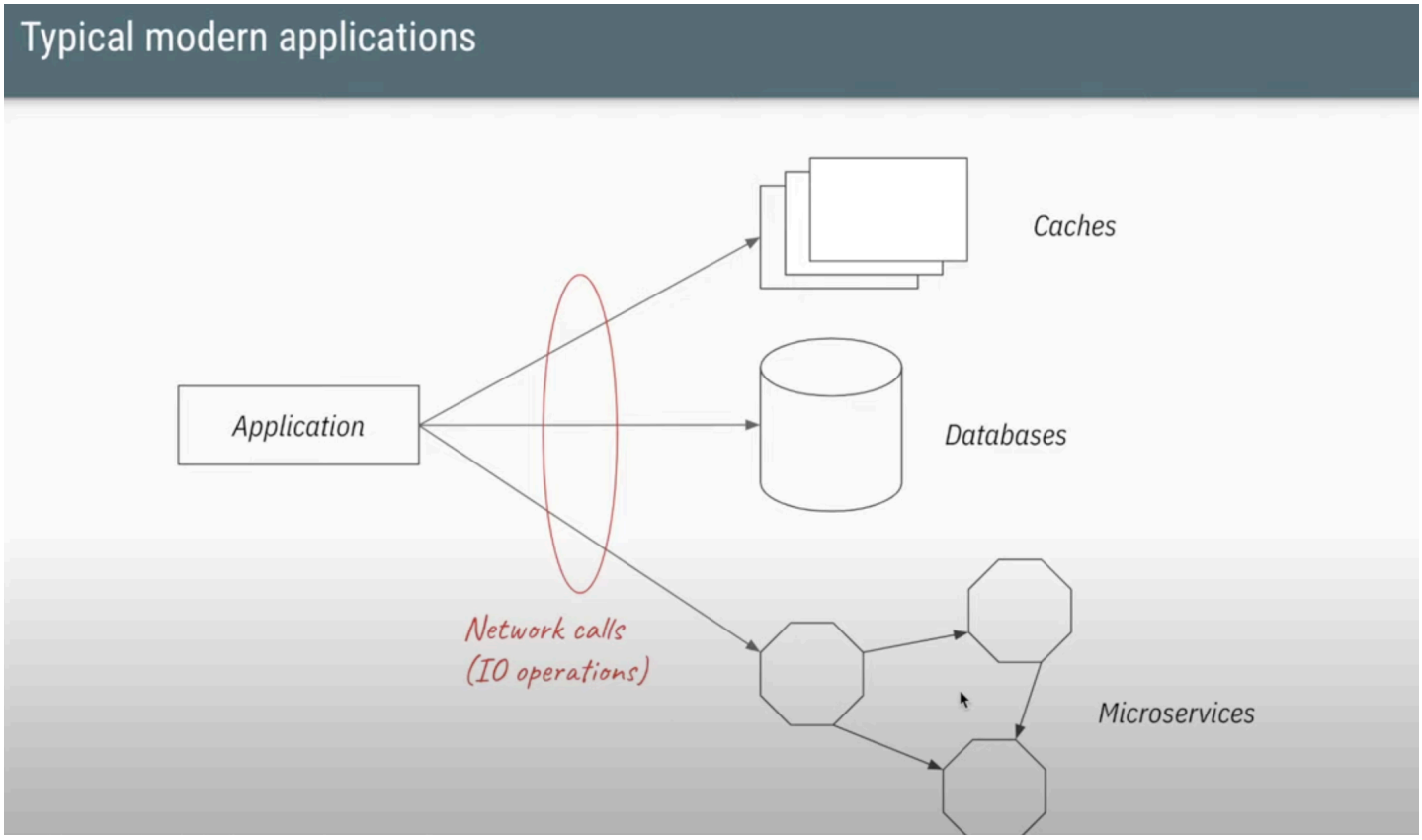


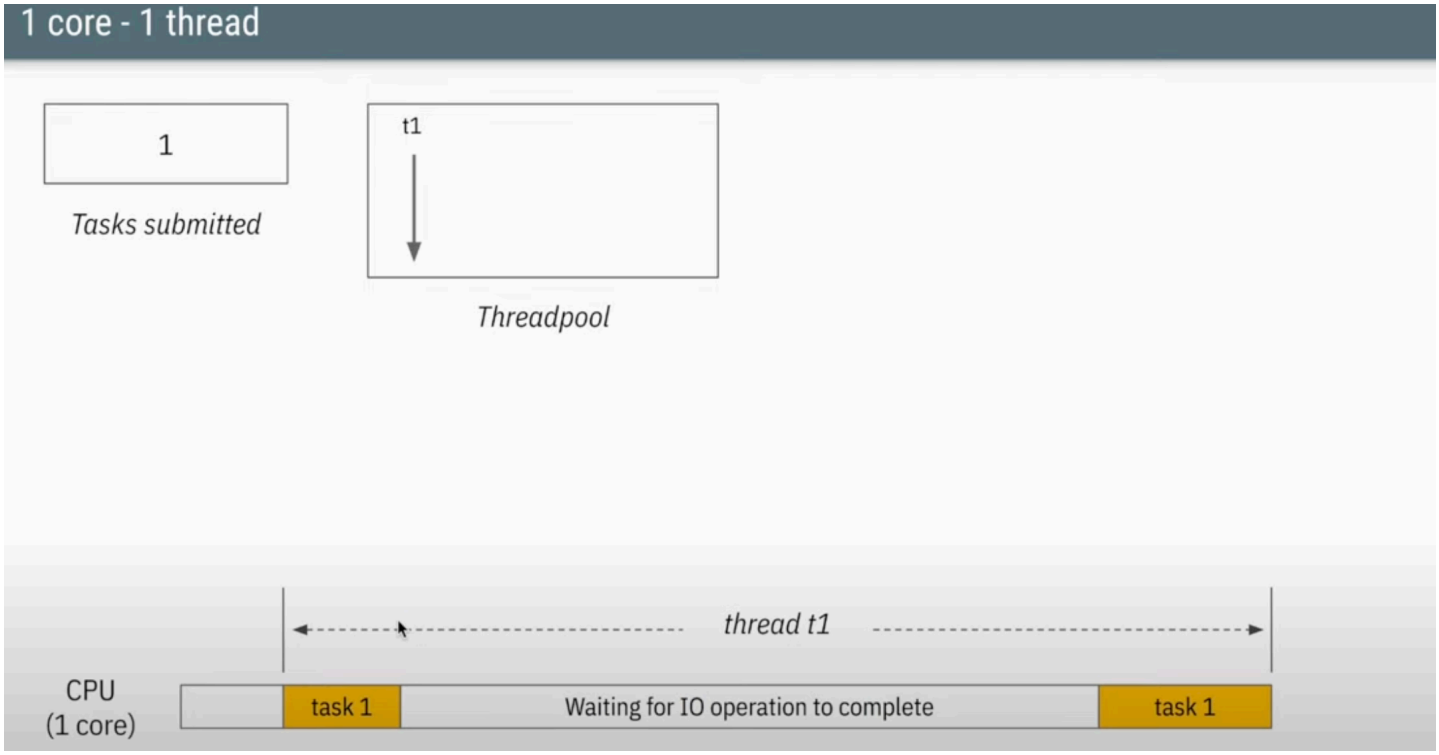
CPU bound tasks

Ideal thread-pool size

=

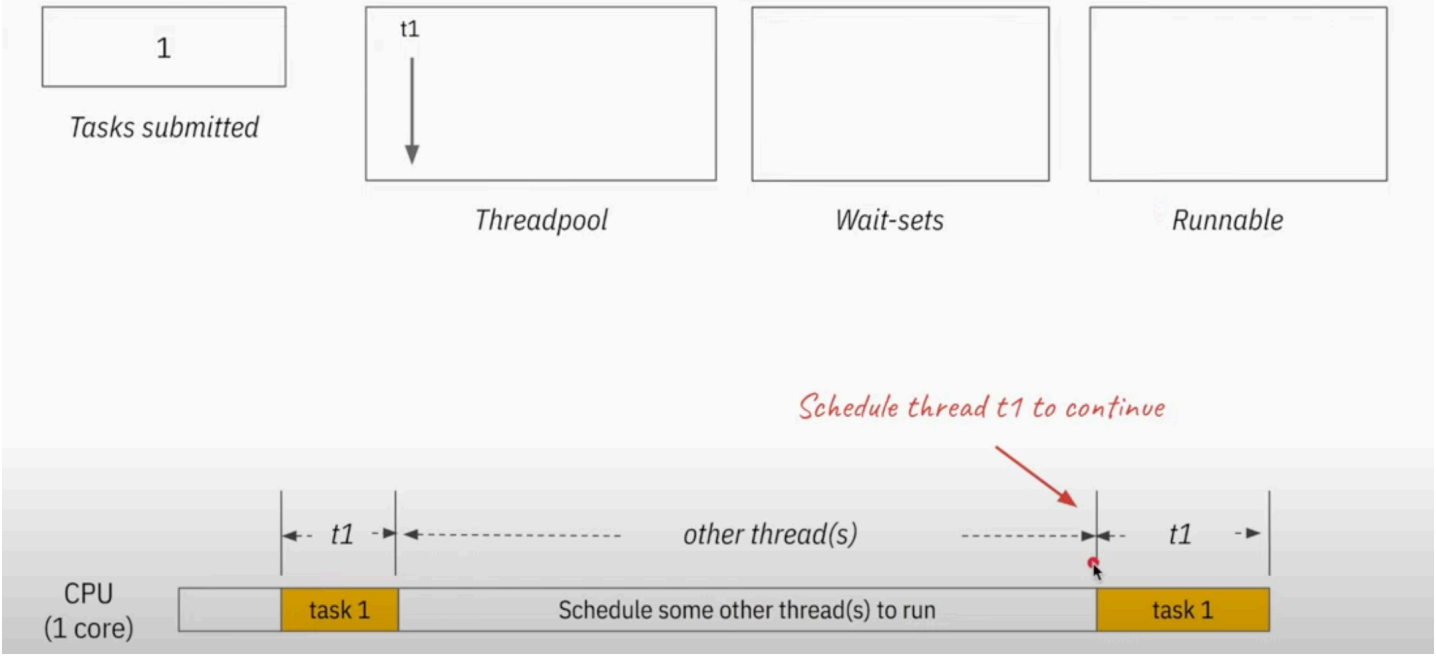
Number of CPU cores



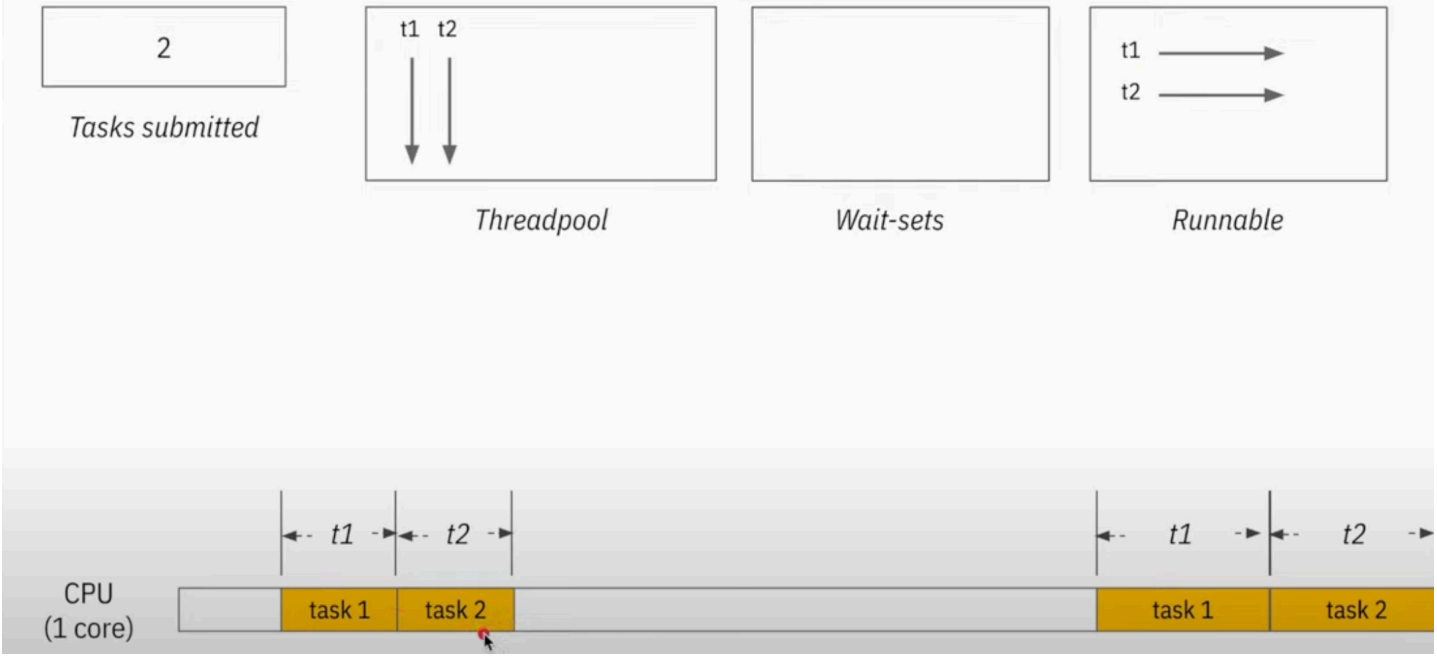


Above is an inefficient use of CPU

1 core - 1 thread



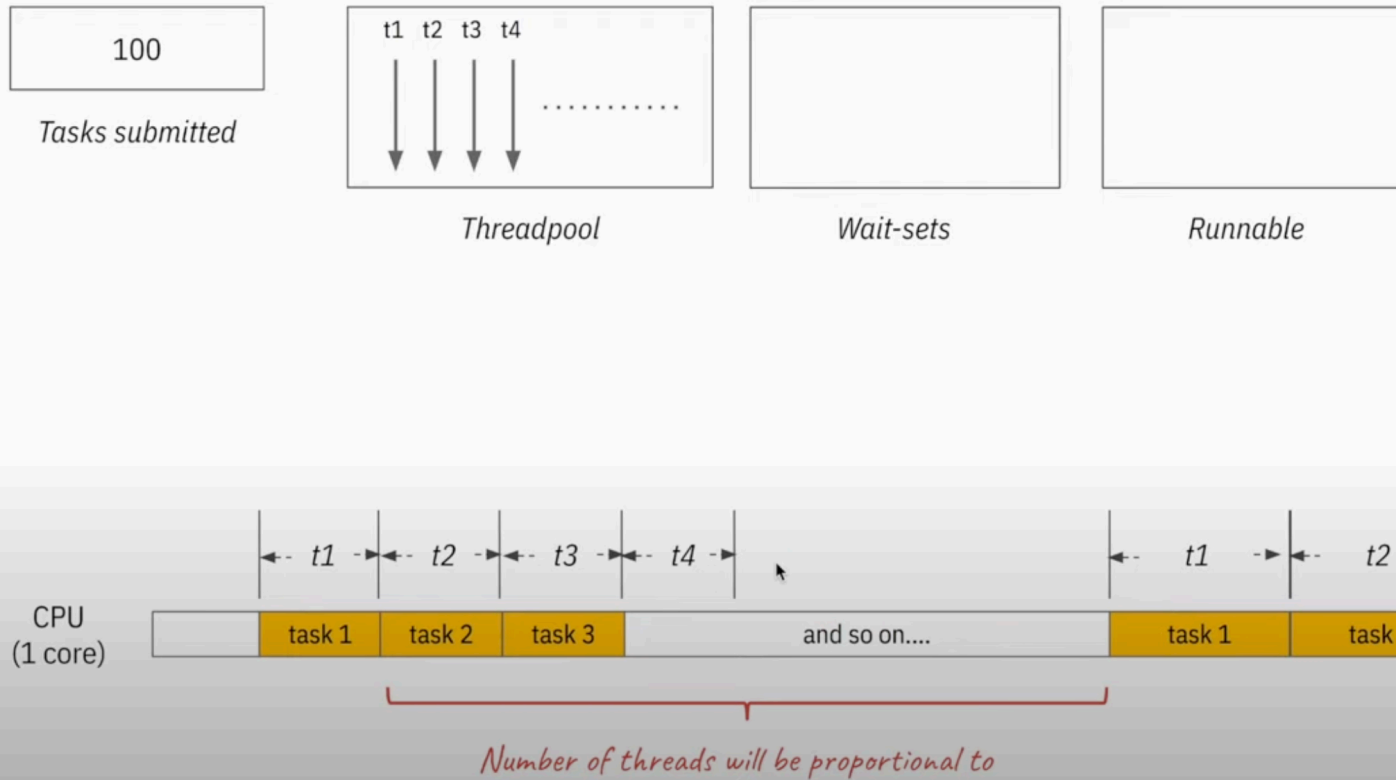
1 core - 1 thread



IO bound tasks

- Even with same number of cores, increasing thread count ***can*** speed up completion

1 core - 1 thread



IO bound tasks

- Number of threads that can be added, depends on how much ***time it takes for one task's IO*** operation to complete

$$\text{Ideal thread count} = \text{Number of cores} \times \left[1 + \frac{\text{Wait time}}{\text{CPU time}} \right]$$

The Formula - CPU bound

$$\text{Ideal thread count} = \text{Number of cores} \times \left[1 + \frac{\text{zero}}{\text{CPU time}} \right]$$

CPU bounds tasks have no wait time, thus this coefficient is zero

The Formula - IO bound

$$\text{Ideal thread count} = \text{Number of cores} \times \left[1 + \frac{\text{Wait time}}{\text{CPU time}} \right]$$

IO bounds tasks have some non-zero wait time

Main factors

- Number of CPU cores
- Task type (CPU / IO)

Other factors

- Are there other applications running on CPU?
- Are there other executor services or threads running in same JVM/application?
- Threads are heavy weight (1-2 MB). Cannot create thousands of them. Even if a task's IO operation allows space for thousand threads.
- Too many threads will also have added complexity/time-taken for thread switching.
- Too many threads also affect data locality (i.e. L1/L2 etc need to be flushed during thread switch).