OS3 - Threads, Executors, Runnables (ignore the order of titles in the curriculum)

Monday, 10 July 2023 6:12 PM

Threads

☆ multiple threads can share same memory.

Let's say we have a CPU bound process with 100 different threads.

We have a processor that supports at most 16 threads at a given time. Now handling 100 threads on this processor would cause lot of Context Switching which is not an desirable scenario.

Let's say we have following tasks to be performed.

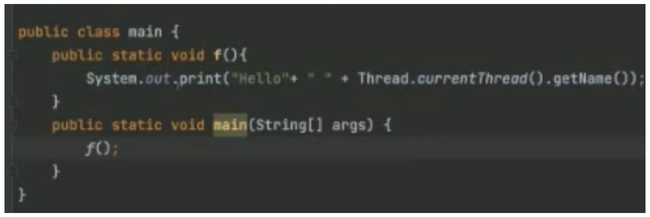
1. Download 10 images using a Python script. → can be multithreaded
2. Perform Kruskal's algo on 10 different testcases → multi threading could lead to too much Context switching
3. Perform something Sequential - print numbers from 1 to 100 in the right order → can be processed on a single thread

When context switching happens, the thread that was currently executing will have it's PCB stored in the RAM before switching to a different thread. Too many threads can consume a lot of memory.

I/O bound processes can be multi-threaded to increase performance in most of the cases.

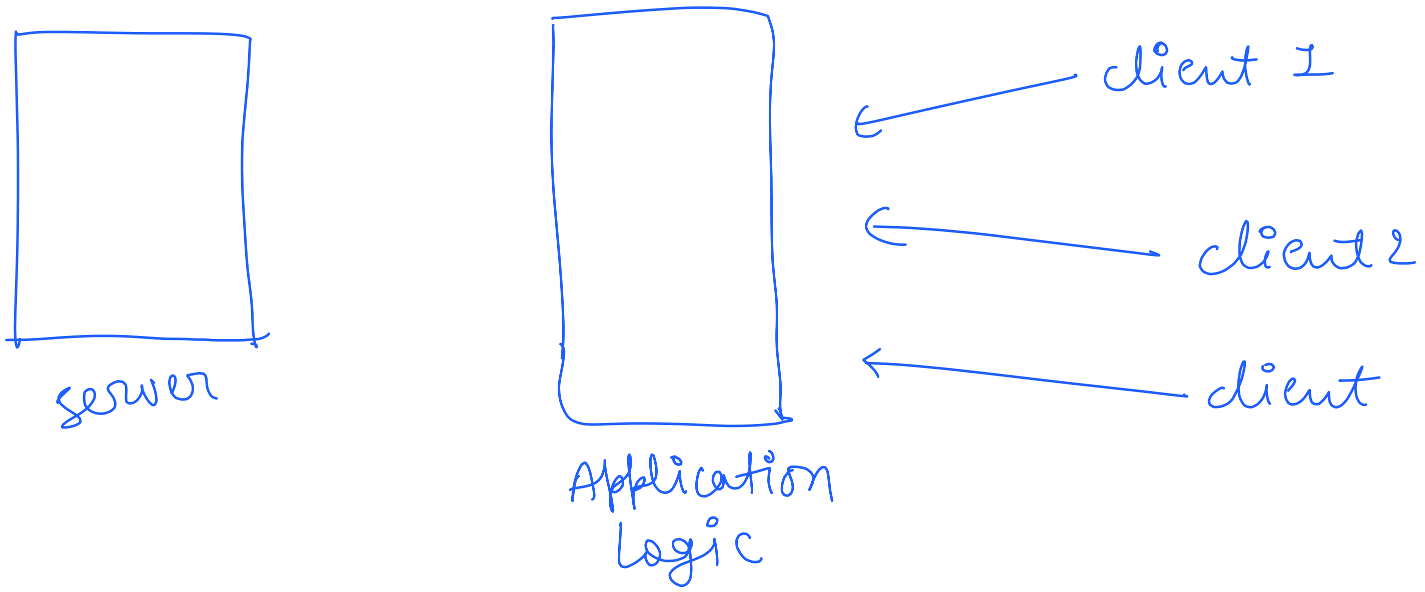
```
import java.util.concurrent.ThreadLocalRandom;
```

Here, since the function f() is being called



```
public class main {
    public static void f(){
        System.out.print("Hello"+ " " + Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        f();
    }
}
```

Here, since the function f() is being called within the Main function, the name of the current thread will remain as 'Main' and not 'f'.



Here, for every client, a dedicated thread is created.
But what happens when we have 1 million users?

1 M users $\longrightarrow$ 1 M threads

→ at most 16 threads can execute

→ this means lot of context switching for 1M

So in order to handle the execution of large number of threads, Executors / Executor Service is used.
It is used in production code.

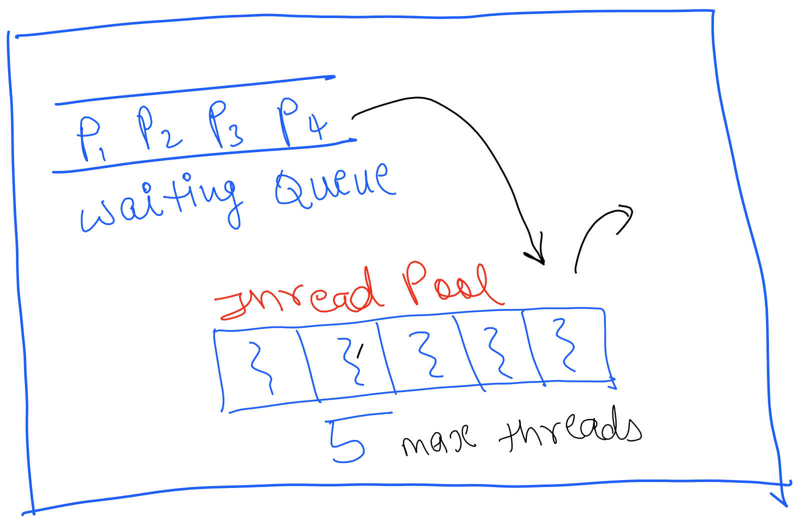
Client

It is the reponsibility of the client to determine which tasks are independent and for what tasks threads need to be created that run parallely.

Executor

Executor determines how to optimally run/schedule these tasks and send them to the CPU

Executor & Thread Pool



So in order to manage the number of threads we can run at once, we can use ExecutorService in Java.

Also, it's not a good idea to create more than one Executor Service.

In Java,
the executor's pre-allocated threads are known as **Workers / Worker Threads**
and
the waiting queue is know as **workerQueue**

It uses existing threads

Assignment:
1. Read about futures in java

public class PrintNumbersWithoutExecutor {

```
public class PrintNumbersWithoutExecutor {
    public static void main(String[] args) {
        for (int i = 1; i <= 100; i++) {
            final int num = i;
            new Thread(() ->
            System.out.println(num)).start();
        }
    }
}
```

2. Create multithreaded Adder/Subtractor
3. Replicate number printer / executor service example.

This will print numbers randomly based on however order the threads are executed internally.

Java:

```
ExecutorService executorService = Executors.newFixedThreadPool(nThreads:10)
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class PrintNumbersWithFixedThreadPool {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        for (int i = 1; i <= 100; i++) {
            final int num = i;
            executorService.submit(() -> System.out.println(num));
        }
        executorService.shutdown();
    }
}
```

```
ExecutorService executorService = Executors.newCachedThreadPool()
```

This creates a thread pool of indefinite size, assigns new tasks to new worker threads on demand, and if there is any existing worker thread that's done it's work, it is re-used again instead of a new worker.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class PrintNumbersWithCachedThreadPool {
    public static void main(String[] args) {
            ExecutorService executorService = Executors.newCachedThreadPool();
            for (int i = 1; i <= 100; i++) {
                final int num = i;
                executorService.submit(() -> System.out.println(num));
            }
            executorService.shutdown();
        }
    }
```

ExecutorService executorService = Executors.newSingleThreadExecutor()

This executes the threads one by one - sequentially.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class PrintNumbersWithSingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        for (int i = 1; i <= 100; i++) {
            final int num = i;
            executorService.submit(() -> System.out.println(num));
        }
        executorService.shutdown();
    }
}
```

Executors :

In the context of multithreading, Executors in Java typically utilize existing threads to execute tasks. Executors provide a high-level interface for managing the execution of tasks in a thread pool. Under the hood, Executors manage a pool of worker threads that are responsible for executing the submitted tasks.

When you create an Executor instance, it typically creates and maintains a fixed number of worker threads in the thread pool. These worker threads are pre-allocated and ready to execute tasks. When you submit a task to the Executor, it assigns the task to one of the available worker threads in the pool for execution. Once a worker thread finishes executing a task, it can be reused to execute another task.

The advantage of using Executors and thread pools is that it allows for efficient thread management and reuse. Instead of creating a new thread for every task, existing threads in the pool are used, reducing the overhead of thread creation and destruction. This can result in improved performance and resource utilization.

It's worth noting that the exact behavior of Executors and thread pools can be customized using different configurations and policies. The specific implementation details can vary based on the Executor type and settings used in your code.

Runnables

In Java, a Runnable is an interface that represents a task or unit of work that can be executed by a thread. The Runnable interface defines a single method called run() without any arguments or return value. When a class implements the Runnable interface and provides an implementation for the run() method, it becomes a "runnable" object.

The run() method of a Runnable encapsulates the code that will be executed by a thread when it is started. It typically contains the main logic or operations that need to be performed by the thread. This could include computations, I/O operations, or any other task that needs to be executed asynchronously.

To execute a Runnable, it needs to be passed to a Thread object. The Thread class provides a constructor that accepts a Runnable as an argument. When the Thread's start() method is called, it creates a new thread and invokes the run() method of the provided Runnable object in that new thread.

By separating the task (implemented as a Runnable) from the thread (represented by a Thread object), Java allows for better separation of concerns and more flexibility in managing and scheduling concurrent tasks.

Additionally, the Java Executor framework, which is built on top of Runnable, provides higher-level abstractions for executing and managing tasks in thread pools, making it easier to handle concurrent programming in Java.

```
import java.util.concurrent.ExecutorService;
```

```java
import java.util.concurrent.Executors;
public class PrintNumbersWithSingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
            for (int i = 1; i <= 100; i++) {
                final int num = i;
                executorService.submit(new NumberPrinter(num));
            }
            executorService.shutdown();
        }
        static class NumberPrinter implements Runnable {
            private final int num;
            public NumberPrinter(int num) {
                this.num = num;
            }

            @Override
            public void run() {
                System.out.println(num);
            }
        }
    }
```