General Overview:

- Today we have processor in our PC's that has many cores that works togather to increase the performance of computations.
- Each core has many threads that runs the code and perform computations.
- When we write a program in Rust (or may be in any other language) that runs on the main thread which means that it'll execute the code line by line or more precisely command by command.
- To execute the next command code must wait for the running command to execute.
- Here comes the multi-threading concept.

Multi-threading:

- Multi-threading is use to perform multi-tasking and to run the CPU-intensive task simultaneously.
- CPU-intensive tasks are those task that donot involve I/O operations. For instant; opening many tabs on a browser.
- Now assume that we have opened 4 tabs on a browser and palying youtube videos on all of them. They'll run all simultaneously on each thread and each thread will also performing some other tasks as well, like fetching data from youtube.com, dispalying ads etc.
- Please see below some coding for understanding of multi-threading:

```
fn talk_drive_eat() { let thread1 = thread::spawn(|| talk()); let thread2 = thread::spawn(|| drive()); let thread3 = thread::spawn(|| eat()); thread1.join().expect("thread1 panicked"); thread2.join().expect("thread2 panicked"); thread3.join().expect("thread3 panicked"); } fn main() { talk_drive_eat(); }
```

- A person is multi-threading because he is driving, eating and talking at the same time which is good but there are some insights which needs to be understand.
- There will be a switching between these three tasks and sharing data between them. If we apply this to our coding then switching and sharing data between many threads will be very difficult and complex task to do. Because we say that multi-threading run code simultaneously but it doesn't. The switching and sharing is soo fast due to our processor speed that we can't feel it.
- Also, when the thread doing nothing and waiting for code to execute it'll utilize system resources.
- Here comes the Async programming to eliminate these issues.

Asynchronous Programming:

- Async is used to run IO intensive tasks and perform multi-tasking with those tasks. It can perform other tasks as weel but not suited for that due to complexity.
- IO intensive tasks are those that includes the input and output operations. For instance; printing out a document, fetching data from a website, scanning a file with scanner and viewing on PC etc.
- In IO tasks system needs to wait alot and during that wait instead of blocking whole thread and utilizing system resources, that thread start working on other tasks.

- Please see below code for better understandign:

```
async fn talk_drive_eat() { let talk = talking(); let drive = driving(); let eat = eating(); join!(talk,drive,eat); }
```

- Above code with the async perform the similar function as the threading but there are some limitations in async as well. Lets disscuss them first.
- Asynchronous programming required extra support from languages or libraries.
- We need to write some extra syntax to define async and we can't write it in every function.
- That is why we only write it when we required to dealt with I/O intensive takss.
- There are executors in async Rust; block_on(), .await() and join!().
- block_on() is use to block the thread and wait for the future. It is used in the synchronous functions.
- Future is the value that we are waiting to return during our call from async function.
- .await() is used in the async function only and do not block the whole thread. It just mark the future as awaiting and let the thread do other tasks.
- join!() is also used in the async functions only and wait for the futures to return and then return to the synchronous function from where async was called.
- Now see it in the code bellow:

```
async fn cellphone_drive() { let cellphone = using().await; drive(cellphone).await; } async fn async_main() { let f1 = cellphone_drive(); let f2 = eat(); let f3 = talking(); futures::join!(f1, f2, f3); } fn main() { block_on(async_main()); }
```

- In above code all three executors has been used. Now we understand the flow of code.
- From main function block_on block the main thread till it return the future. Async_main will start to execute.
- cellphone_drive async function will be called first from async_main in a thread where driving will not performed while using cellphone.
- during above calling of function eating and talking will be executing on other threads without any interuption and when the cellphone_drive will return its future join executore join these three and get the futures from all three then return the future from async_main to the main synchronous function and thread finished there work.

Conclusion:

Multi-threading is used to perform tasks on different threads and increase the utilization of the processor while async is used to perform multiple taks on single thread so that we can utilize the processor performance to its maximum while doing I/O intensive tasks.