Round Robin scheduling algorithm and threads

Round Robin

> Round-robin (RR) is one of the algorithms employed by process and network schedulers in computing. As the term is generally used, time slices (also known as time quanta) are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is simple, easy to implement, and starvation-free. Round-robin scheduling can be applied to other scheduling problems, such as data packet scheduling in computer networks.

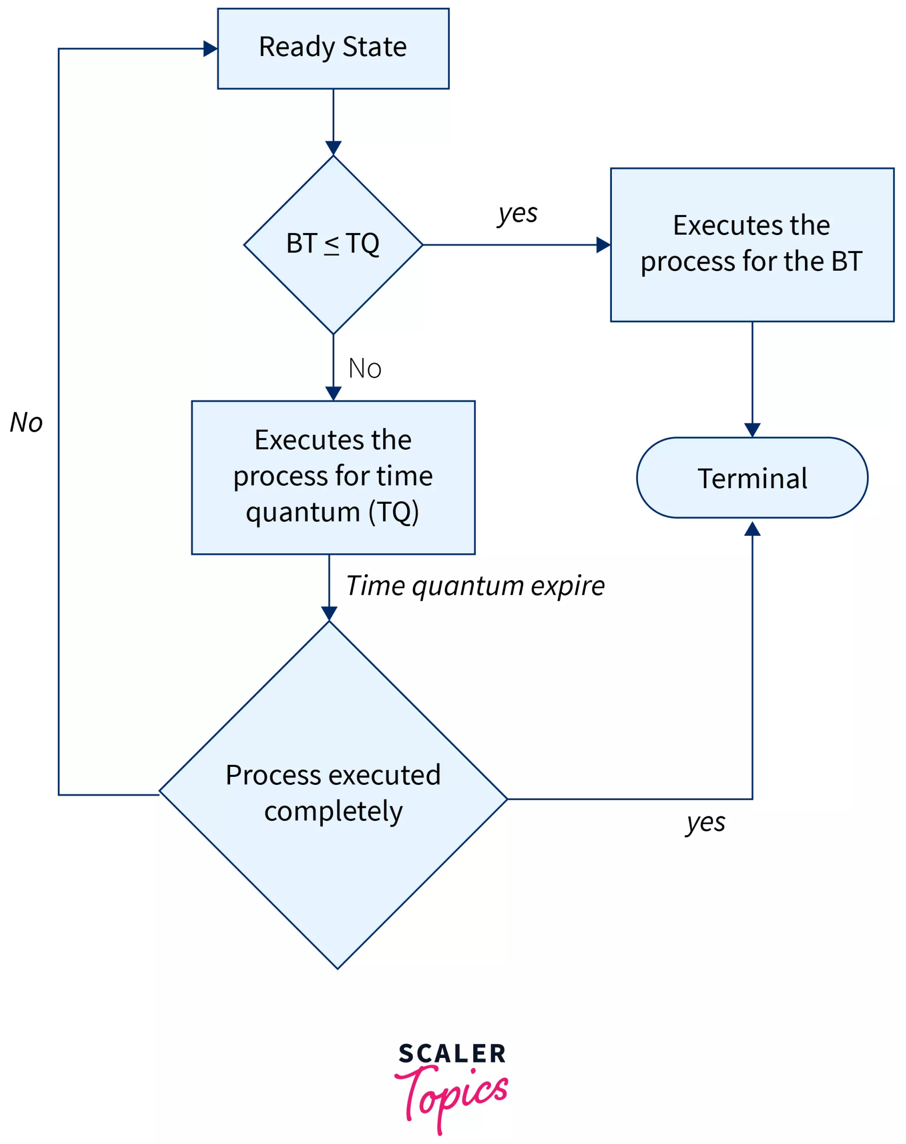
One of the major problems with SRTF is that it can lead to starvation. This is because a process with a long burst time can be starved of CPU time. Often the requirement is to have a scheduling algorithm that is fair to all processes so that all processes are progressing and they have lower waiting times. This is where Round Robin scheduling comes in.

Round Robin scheduling is a preemptive scheduling algorithm. This means that the scheduler can preempt a process and give the CPU to another process.

This is done after a time slice. The time slice is the amount of time a process is allowed to run before it is preempted. The time slice is usually a small value like 10ms. This means that a process is given 10ms of CPU time and then it is preempted and another process is given 10ms of CPU time. This is repeated until all processes have been given CPU time. This is called Round Robin scheduling because the CPU is given to each process in a round robin fashion.

Algorithm

- If no process is running, pick the next process from the queue and run it.
- The process runs either till the minimum of the time slice or the burst time. `RunnningTime = min(timeSlice, burstTime)`
- If the process is not finished, put it back in the queue.
- Repeat the above steps till the queue is empty.


Ready State
BT ≤ TQ
yes
Executes the process for the BT
No
Executes the process for time quantum (TQ)
Terminal
No
Time quantum expire
Process executed completely
yes
SCALER Topics

Example

Let us consider the following processes with their arrival time and burst time.

Process	Arrival Time	Burst Time
P1	0	5
P2	1	6
P3	2	3
P4	3	1
P5	4	5
P6	6	4

Let us consider our time slice to be 4 units.

At t=0

- P1 arrives and is added to the queue.
- Since no process is running, P1 is picked and is given 4 units of CPU time.

At t=1

- P2 arrives and is added to the queue.
- P1 is still running and has 3 unit of CPU time left.

At t=2

- P3 arrives and is added to the queue.
- P1 is still running and has 2 unit of CPU time left.

At t=3

- P4 arrives and is added to the queue.
- P1 is still running and has 1 unit of CPU time left.

The queue now looks like this:

Process	Remaining Time
P2	6
P3	3
P4	1

At t=4

- P5 arrives and is added to the queue.
- The time slice for P1 is over.
- P2 is picked and is given 4 units of CPU time.
- P1 is added to the queue with 1 unit of CPU time left.

The queue now looks like this:

Process	Remaining Time
P3	3
P4	1
P5	5
P1	1

At t=6

- P6 arrives and is added to the queue.
- P2 is still running and has 2 units of CPU time left.

At t=8

- The time slice for P2 is over.
- P3 is picked and is given 4 units of CPU time.
- P2 is added to the queue with 2 units of CPU time left.

The queue now looks like this:

Process	Remaining Time
P4	1
P5	5
P1	1
P6	4
P2	2

At t=11

- P3 has finished running.
- P4 is picked and has 1 unit of CPU time left.

The queue now looks like this:

Process	Remaining Time
P5	5
P1	1
P6	4
P2	2

At t=12

- P4 has finished running.
- P5 is picked and is given 4 units of CPU time.

The queue now looks like this:

Process	Remaining Time
P1	1
P6	4
P2	2

At t=16

- The time slice for P5 is over.
- P1 is picked and has 1 unit of CPU time left.
- P5 is added to the queue with 1 unit of CPU time left.

At t=17

- P1 has finished running.
- P6 is picked and is given 4 units of CPU time.

The queue now looks like this:

Process	Remaining Time
P2	2
P5	1

At t=21

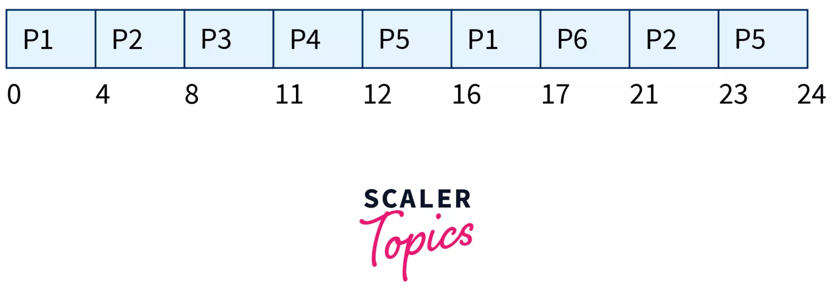
- P6 has finished running.
- P2 is picked up and has 2 units of CPU time left.

At t=23

- P2 has finished running.
- P5 is picked up and has 1 unit of CPU time left.

At t=24

- P5 has finished running.
- No more processes are left in the queue.



Advantages

- This round robin algorithm offers starvation-free execution of processes.
- Each process gets equal priority and fair allocation of CPU.
- It is easily implementable on the system because round robin scheduling in os doesn't depend upon burst time.

Disadvantages

- The waiting time and response time are higher due to the short time slot.
- Lower time quantum results in higher context switching.
- We cannot set any special priority for the processes.

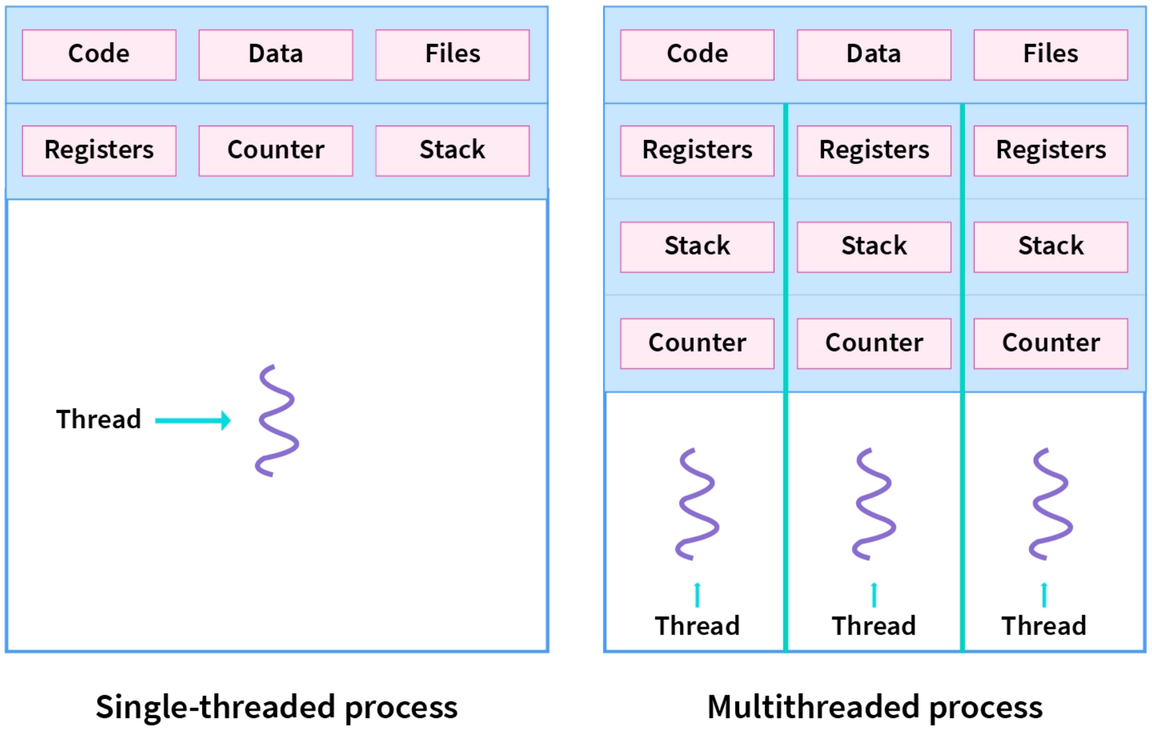
Threads

> A thread is a lightweight process. It is a unit of execution within a process. A process can have multiple threads. Each thread has its own program counter, stack, and registers. Threads share the same address space. This means that all threads in a process can access the same memory. This is different from processes where each process has its own address space.

Often, a process needs to perform multiple tasks at the same time. For example, a web browser needs to download a file and display a web page at the same time. Creating a new process for each task is expensive. This is because creating a new process requires a lot of resources.

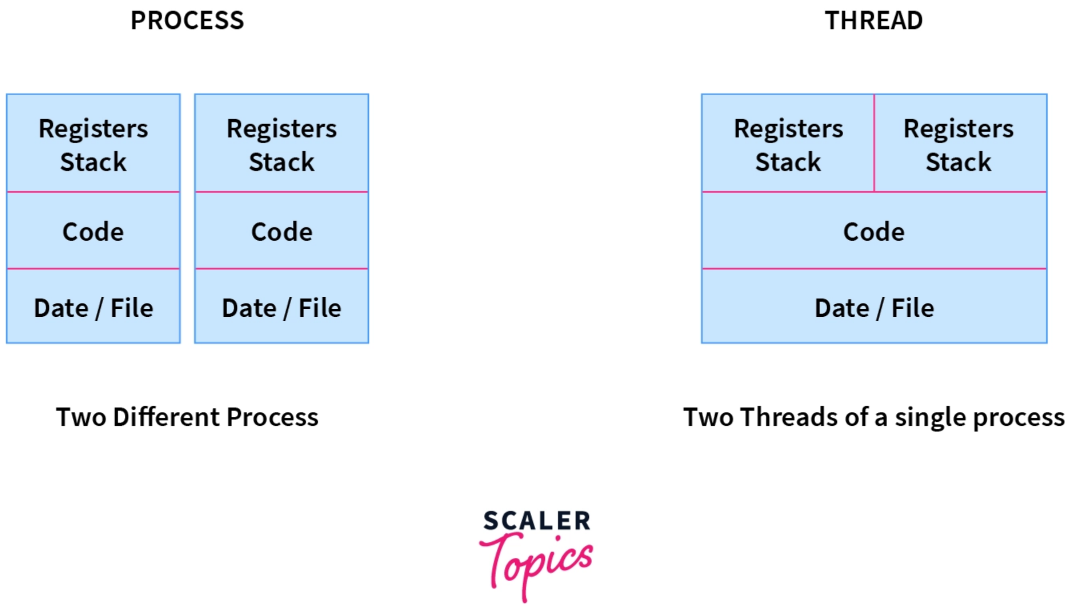
Threads are used to solve this problem. Threads are used to perform multiple tasks within a process. This is done by sharing the same address space. This means that all threads in a process can access the same memory. This is different from processes where each process has its own address space.

Thread is a sequential flow of tasks within a process. Threads in OS can be of the same or different types. Threads are used to increase the performance of the applications. Each thread has its own program counter, stack, and set of registers. But the threads of a single process might share the same code and data/file. Threads are also termed as lightweight processes as they share common resources.



Thread vs Process

Process	Thread
Processes use more resources and hence they are termed as heavyweight processes.	Threads share resources and hence they are termed as lightweight processes.
Creation and termination times of processes are slower.	Creation and termination times of threads are faster compared to processes.
Processes have their own code and data/file.	Threads share code and data/file within a process.
Communication between processes is slower.	Communication between threads is faster.
Context Switching in processes is slower.	Context switching in threads is faster.
Processes are independent of each other.	Threads, on the other hand, are interdependent. (i.e they can read, write or change another thread's data)
Eg: Opening two different browsers.	Eg: Opening two tabs in the same browser.



Concurrency vs Parallelism

- Concurrent - At the same time, but not necessarily at the same instant. It is possible for multiple threads to be at different stages of execution at the same time but not being processed together. A single core CPU can only execute one thread at a time. But it can switch between threads very quickly. This is called context switching. This is how concurrency is achieved. A single core CPU can have concurrency but not parallelism.
- Parallel - At the same time and at the same instant. It is possible for multiple threads to be at different stages of execution at the same time and being processed together. A single core CPU cannot achieve parallelism. It can only achieve concurrency. A multi-core CPU can achieve both concurrency and parallelism.

Using threads in Java

In Java, we can create a thread by extending the Thread class or by implementing the Runnable interface. The Thread class is a subclass of the Object class. It implements the Runnable interface. The Runnable interface has a single method called run(). This method is called when the thread is started.

```
class NewThread implements Runnable {
    @Override
    public void run() {
        // Code to be executed by the thread
    }
}
```

We can create a new thread by creating an object of the NewThread class and passing it to the Thread class constructor. The Thread class constructor takes a Runnable object as an argument. This Runnable object is the thread that we want to create.

```
NewThread newThread = new NewThread();
Thread thread = new Thread(newThread);
```

To run the thread, we call the start() method on the Thread object. This method calls the run() method of the Runnable object. The run() method is executed by the thread.

```
thread.start();
```

Number printer

Problem Statement

- Create a new thread that prints the numbers from 1 to 10.

Solution

```
class NumberPrinter implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        NumberPrinter numberPrinter = new NumberPrinter();
        Thread thread = new Thread(numberPrinter);
        thread.start();
    }
}
```

Problem Statement 2

- Print the numbers from 1 to 10 where each number is printed by a different thread.

Solution

```java
class NumberPrinter implements Runnable {
    private int number;

    public NumberPrinter(int number) {
        this.number = number;
    }

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

```java
public class Main {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            NumberPrinter numberPrinter = new NumberPrinter(i);
            Thread thread = new Thread(numberPrinter);
            thread.start();
        }
    }
}
```

Assignment

- Create a count class that has a count variable.
- Create two different classes `Adder` and `Subtractor`.
- Accept a count object in the constructor of both the classes.
- In `Adder`, iterate from 1 to 100 and increment the count variable by 1 on each iteration.
- In `Subtractor`, iterate from 1 to 100 and decrement the count variable by 1 on each iteration.
- Print the final value of the count variable.
- What would the ideal value of the count variable be?

- What is the actual value of the count variable?
- Try to add some delay in the `Adder` and `Subtractor` classes using inspiration from the code below. What is the value of the count variable now?

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Reading List

- Web Browser architecture