OS Final Project

Helia Ghorbani - 9824353

Sadaf Abedi - 9823453

Problem

- Multi-core processor scheduler
    - Number of processes: M
    - Each process has:
        - Arrival time
        - Burst time
        - Core-needed
    - Number of CPU cores: N
    - Scheduler:
        - Algorithm: FCFS
        - Non-preemptive
    - Processes:
        - Random number by a random duration
        - Using ThreadPool
    - Processor:
        - Number of cores
        - Method: CPU cores allocator
        - Manage critical section by using lock
- Important points:
    - If all cores are busy, print “processor is busy”.

- Report system status every 5 seconds:
    - Number of free cores
    - Number of busy cores
    - Run time duration
    - Number of running processes
    - Number of processes in the ready queue

Overview

we consider N cores and M processes and N threads.

Each core is a single thread that works synchronously. Initially, all cores are waiting for processes to arrive.

Whenever processes arrive, a core is assigned to processes and a core is decremented from available cores. After serving the processes, core will be free for another process. The core then checks if there are any other processes present in threads or not. If present, then he calls the processes based on a first come first serve basis.

until N cores are taken, we process N processes because each core can do one process.

if the number of processes is greater than the number of cores and no available/free threads exist in our cores, processes are waiting in the queue.

Each process is a separate thread and these threads are executed on multiple cores using Executor service. processes enter the processor and check the status of cores (i.e. if cores are busy or free). If the core is free, then the process takes it And if core is already busy, then process goes to waiting part. If processes find the vacant place to wait, then it acquires that place (which in turn decrements waiting, place count). If there is no place available in waiting area, then that process leaves the processor.

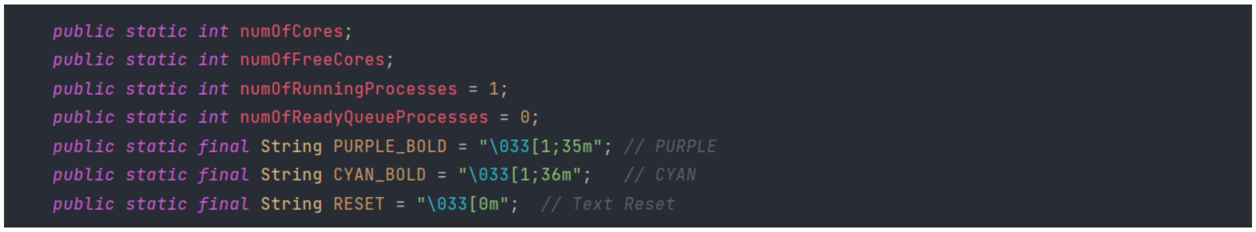
These processes are randomly assigned a time to run and are repeated at different time intervals according to the scheduling algorithm according to the number of cores required.

- ❖ Executor service: Executor service is used to execute the task concurrently that are assigned by threads. ThreadPoolExecutor implementation of Executor service is used in the program. After all threads are executed, the executor is shutdown.
- ❖ Synchronized block: Synchronized blocks are used in program to avoid race condition. This block is synchronized on a list of processes so that this list is accessed by only one thread at a time.

Code Explanation

- **Main**

Parameters



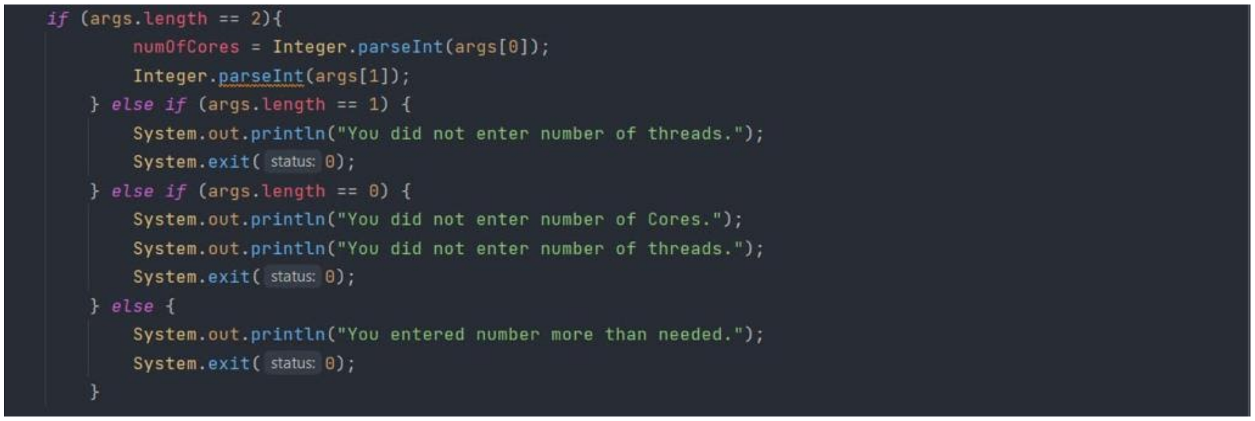
```
public static int numOfCores;
public static int numOfFreeCores;
public static int numOfRunningProcesses = 1;
public static int numOfReadyQueueProcesses = 0;
public static final String PURPLE_BOLD = "\033[1;35m"; // PURPLE
public static final String CYAN_BOLD = "\033[1;36m";   // CYAN
public static final String RESET = "\033[0m";  // Text Reset
```

To generate processes at intervals of at random intervals, with mean M and standard deviation SD. processGenerator object has been created and started with Thread object.

processGenerator class has been developed by use of Dynamic Dispatch(Runtime Polymorphism) whereas, Reference of Random class and object of child class, in order to use nextGaussian function.

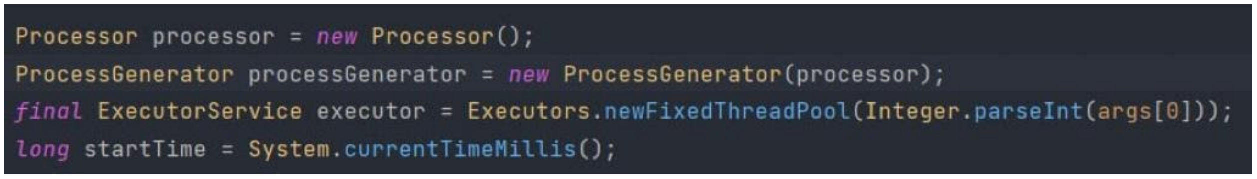
processGenerator class needs an object of Processor class because it has a method of aligning processes in a queue.

This Block of the main method is for Input Validation for a number of cores and number of running threads.

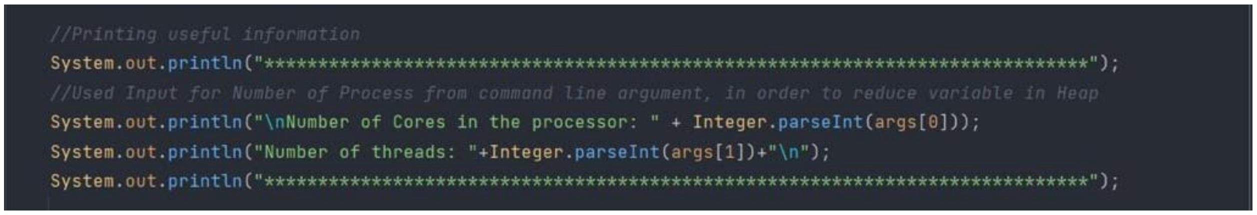


```
if (args.length == 2){
        numOfCores = Integer.parseInt(args[0]);
        Integer.parseInt(args[1]);
    } else if (args.length == 1) {
        System.out.println("You did not enter number of threads.");
        System.exit( status: 0);
    } else if (args.length == 0) {
        System.out.println("You did not enter number of Cores.");
        System.out.println("You did not enter number of threads.");
        System.exit( status: 0);
    } else {
        System.out.println("You entered number more than needed.");
        System.exit( status: 0);
    }
```

This block is for Processor and processGenerator object creation and accessing



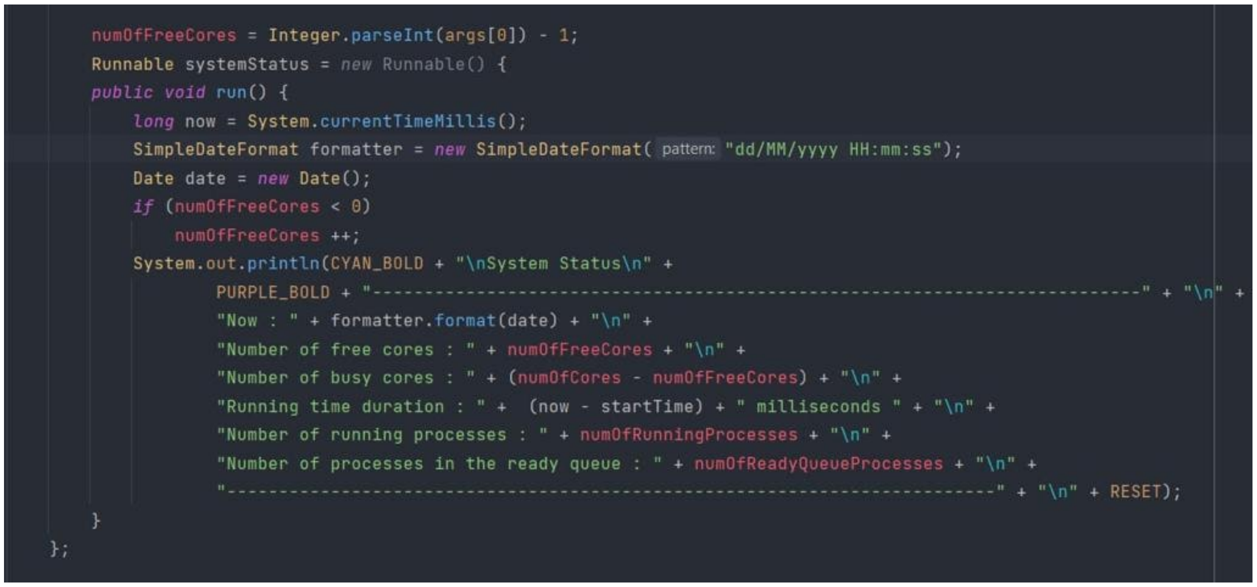
```
Processor processor = new Processor();
ProcessGenerator processGenerator = new ProcessGenerator(processor);
final ExecutorService executor = Executors.newFixedThreadPool(Integer.parseInt(args[0]));
long startTime = System.currentTimeMillis();
```



```
//Printing useful information
System.out.println("****************************************************************************");
//Used Input for Number of Process from command line argument, in order to reduce variable in Heap
System.out.println("\nNumber of Cores in the processor: " + Integer.parseInt(args[0]));
System.out.println("Number of threads: "+Integer.parseInt(args[1])+"\n");
System.out.println("****************************************************************************");
```

available core for Parallel processing using Multi-core using available core in Executor Services for Multi Core.

This part is specifies various program details including the number of free cores, number of busy cores, running time duration, number of running processes and number of processes in the ready queue.

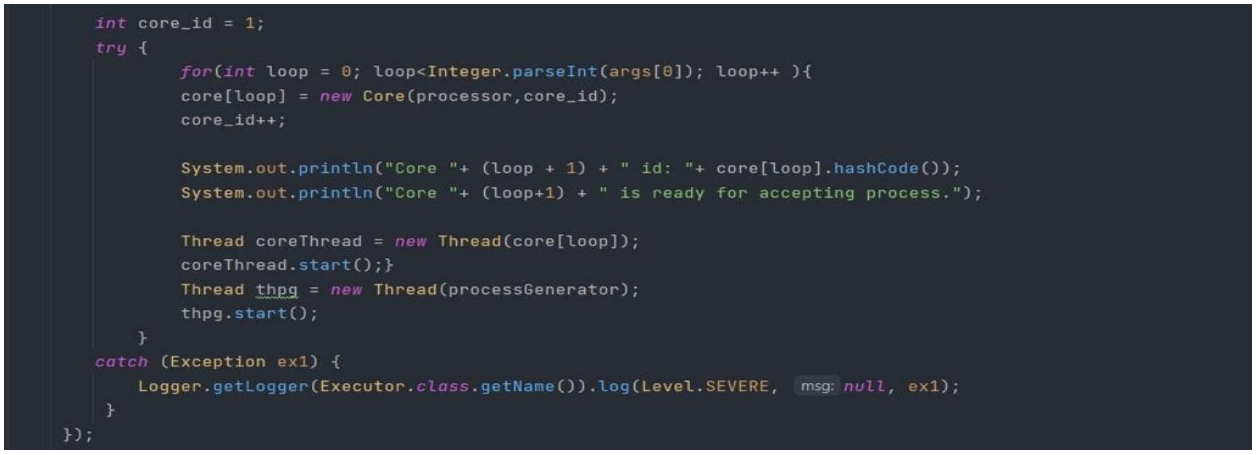


```
    numOfFreeCores = Integer.parseInt(args[0]) - 1;
    Runnable systemStatus = new Runnable() {
    public void run() {
        long now = System.currentTimeMillis();
        SimpleDateFormat formatter = new SimpleDateFormat( pattern: "dd/MM/yyyy HH:mm:ss");
        Date date = new Date();
        if (numOfFreeCores < 0)
            numOfFreeCores ++;
        System.out.println(CYAN_BOLD + "\nSystem Status\n" +
                PURPLE_BOLD + "--------------------------------------------------------------------------" + "\n" +
                "Now : " + formatter.format(date) + "\n" +
                "Number of free cores : " + numOfFreeCores + "\n" +
                "Number of busy cores : " + (numOfCores - numOfFreeCores) + "\n" +
                "Running time duration : " +  (now - startTime) + " milliseconds " + "\n" +
                "Number of running processes : " + numOfRunningProcesses + "\n" +
                "Number of processes in the ready queue : " + numOfReadyQueueProcesses + "\n" +
                "--------------------------------------------------------------------------" + "\n" + RESET);
    }
};
```

This block is for starting executor; the core counter starts from 1.

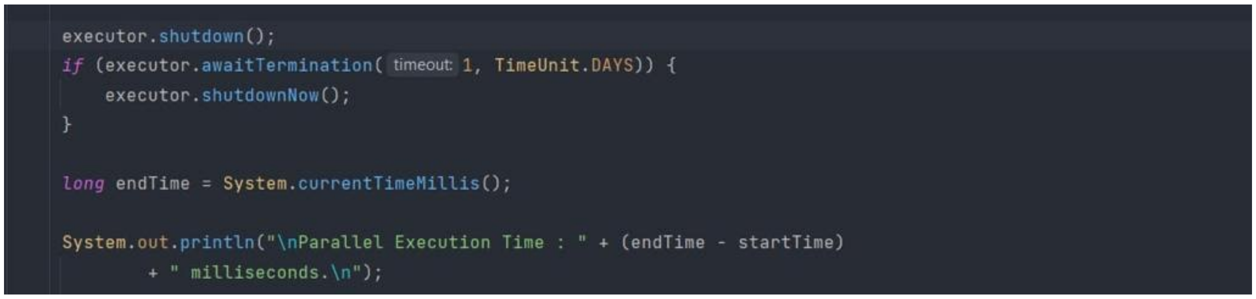
Below Loop will generate N core and Start Thread of each core.

Variable core_id has been used to keep count of Cores has been created dynamically.



```
    int core_id = 1;
    try {
            for(int loop = 0; loop<Integer.parseInt(args[0]); loop++ ){
            core[loop] = new Core(processor,core_id);
            core_id++;

            System.out.println("Core "+ (loop + 1) + " id: "+ core[loop].hashCode());
            System.out.println("Core "+ (loop+1) + " is ready for accepting process.");

            Thread coreThread = new Thread(core[loop]);
            coreThread.start();}
            Thread thpg = new Thread(processGenerator);
            thpg.start();
        }
    catch (Exception ex1) {
        Logger.getLogger(Executor.class.getName()).log(Level.SEVERE,  msg: null, ex1);
    }
});
```

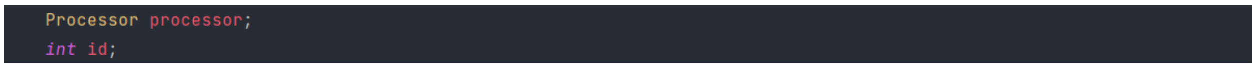
This part of the code stops executor and prints execution time of the program.



```
executor.shutdown();
if (executor.awaitTermination( timeout: 1, TimeUnit.DAYS)) {
    executor.shutdownNow();
}

long endTime = System.currentTimeMillis();

System.out.println("\nParallel Execution Time : " + (endTime - startTime)
        + " milliseconds.\n");
```

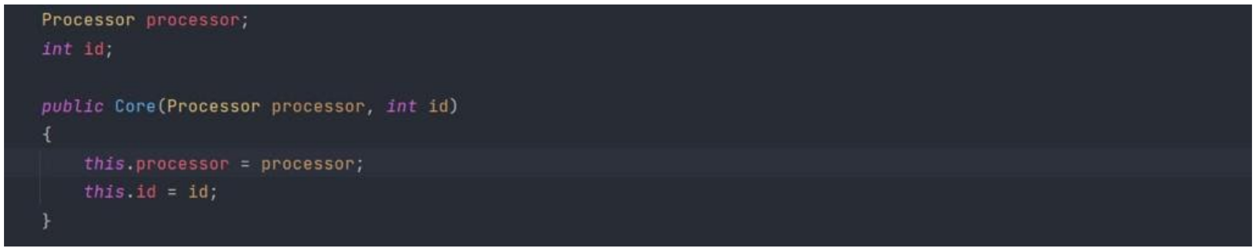
- **Core**

Parameters



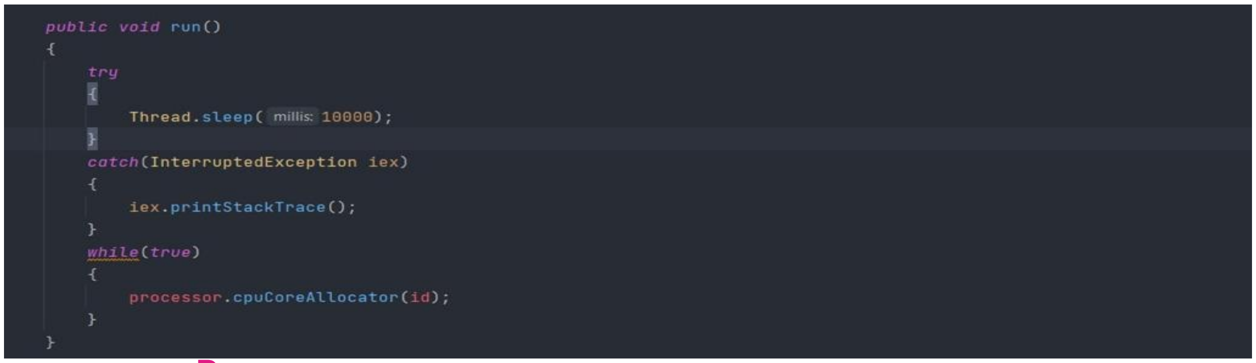
```
Processor processor;
int id;
```

We wrote a class for core and in this code section take parameters and create the constructor of it.



```
Processor processor;
int id;

public Core(Processor processor, int id)
{
    this.processor = processor;
    this.id = id;
}
```

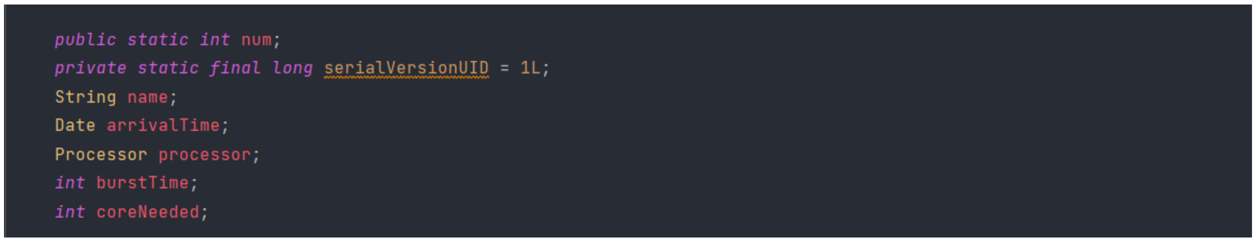
In this section processor can call the specific core by its id.



```
public void run()
{
    try
    {
        Thread.sleep( millis: 10000);
    }
    catch(InterruptedException iex)
    {
        iex.printStackTrace();
    }
    while(true)
    {
        processor.cpuCoreAllocator(id);
    }
}
```

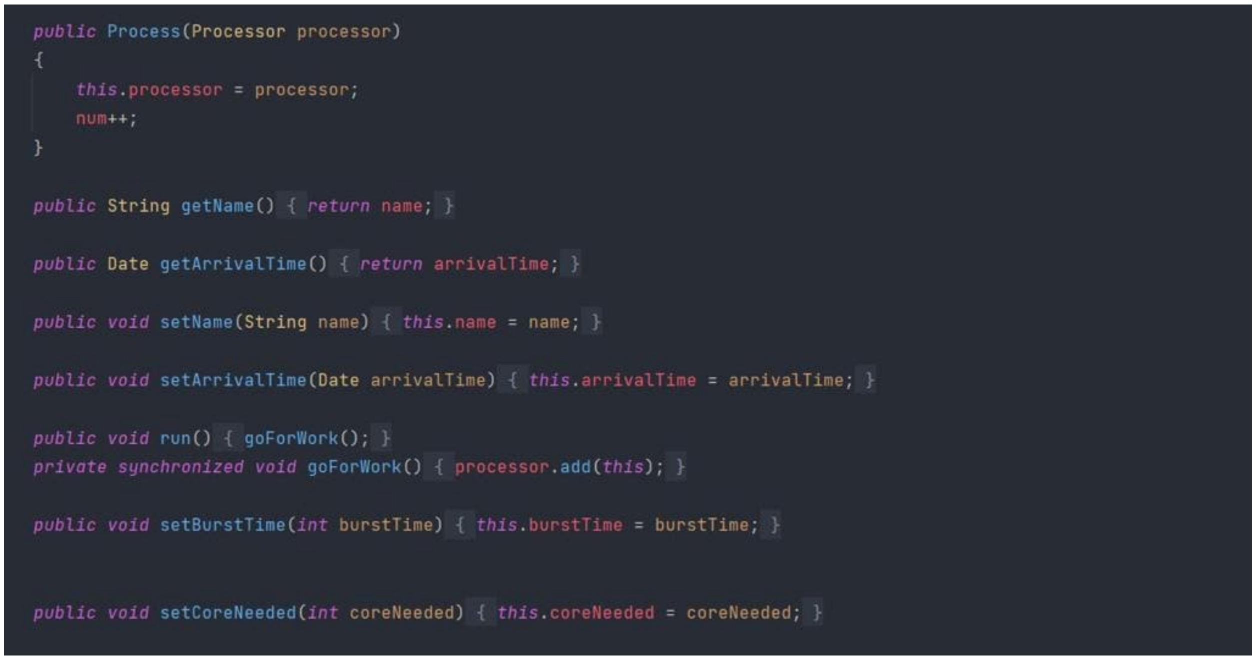
- **Process**

Parameters



```
public static int num;
private static final long serialVersionUID = 1L;
String name;
Date arrivalTime;
Processor processor;
int burstTime;
int coreNeeded;
```

In this class, we just get the parameters and create the constructor for it.



```
public Process(Processor processor)
{
    this.processor = processor;
    num++;
}

public String getName() { return name; }

public Date getArrivalTime() { return arrivalTime; }

public void setName(String name) { this.name = name; }

public void setArrivalTime(Date arrivalTime) { this.arrivalTime = arrivalTime; }

public void run() { goForWork(); }
private synchronized void goForWork() { processor.add(this); }

public void setBurstTime(int burstTime) { this.burstTime = burstTime; }


public void setCoreNeeded(int coreNeeded) { this.coreNeeded = coreNeeded; }
```

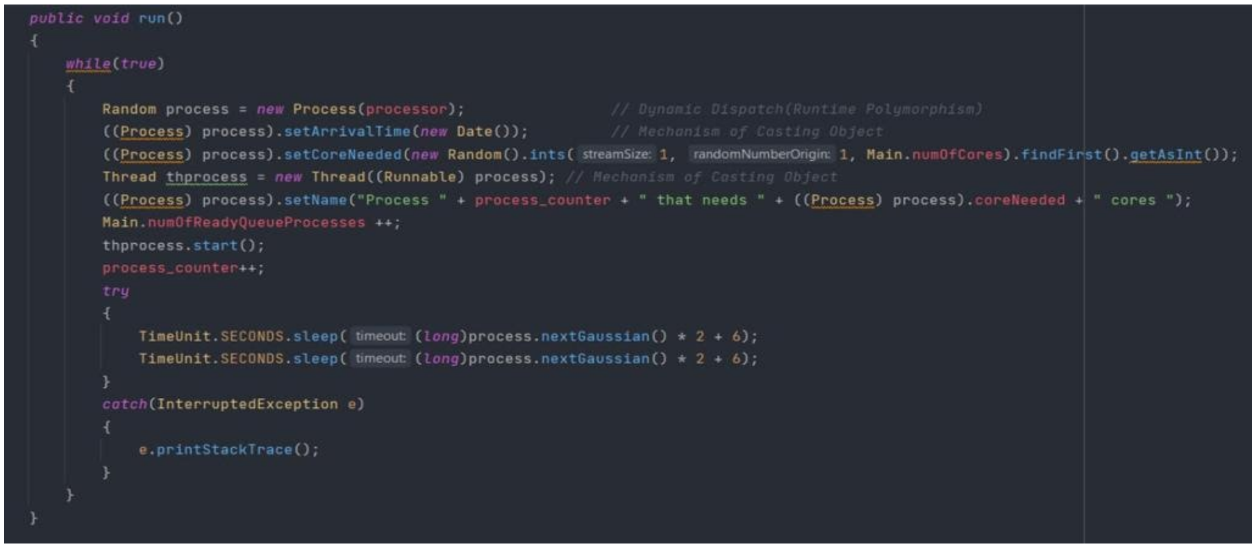
• Process Generator

Parameters

```
Processor processor;
public int process_counter = 1;
```

This section lets the new processes are generated and requirements to be adjusted.

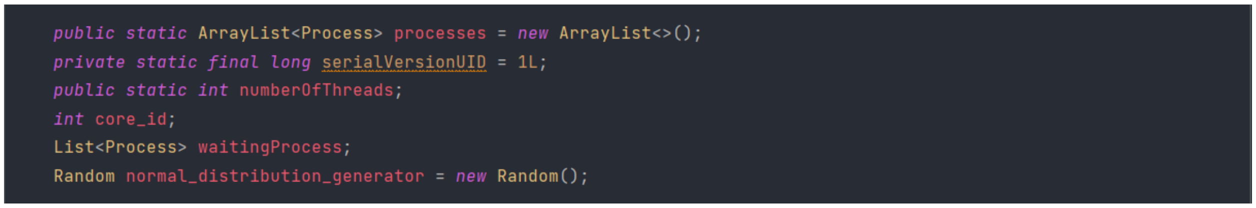
Since nextGaussian() is method of Random Class, so in order to use this method with process Class Instance, thus created reference variable of Random Class(Parent Class) and Object of process Class(Child Class) i.e Dynamic Dispatch mechanism(Runtime Polymorphism).



```
public void run()
{
    while(true)
    {
        Random process = new Process(processor);              // Dynamic Dispatch(Runtime Polymorphism)
        ((Process) process).setArrivalTime(new Date());        // Mechanism of Casting Object
        ((Process) process).setCoreNeeded(new Random().ints( streamSize: 1,  randomNumberOrigin: 1, Main.numOfCores).findFirst().getAsInt());
        Thread thprocess = new Thread((Runnable) process); // Mechanism of Casting Object
        ((Process) process).setName("Process " + process_counter + " that needs " + ((Process) process).coreNeeded + " cores ");
        Main.numOfReadyQueueProcesses ++;
        thprocess.start();
        process_counter++;
        try
        {
            TimeUnit.SECONDS.sleep( timeout: (long)process.nextGaussian() * 2 + 6);
            TimeUnit.SECONDS.sleep( timeout: (long)process.nextGaussian() * 2 + 6);
        }
        catch(InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

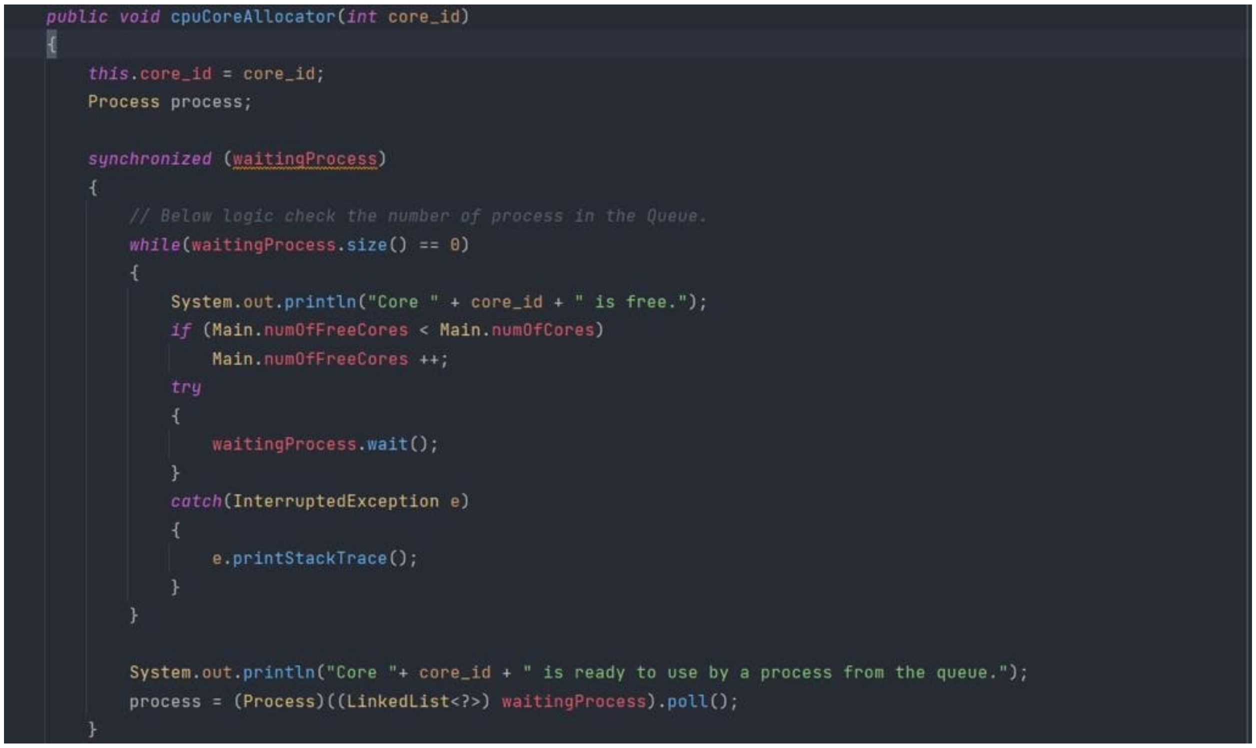
- **Processor**

Parameters



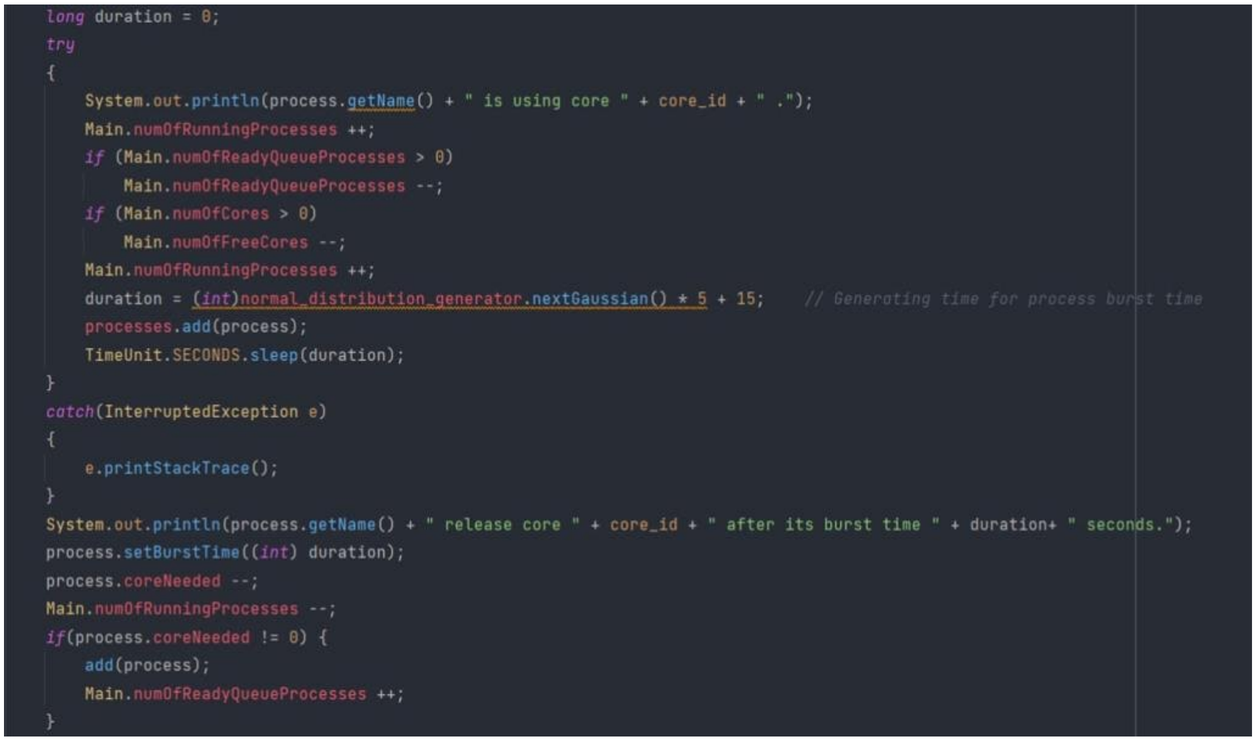
```java
public static ArrayList<Process> processes = new ArrayList<>();
private static final long serialVersionUID = 1L;
public static int numberOfThreads;
int core_id;
List<Process> waitingProcess;
Random normal_distribution_generator = new Random();
```

This code section is the first part of scheduling that is specifies which core is free and which is ready to use by a valid process in the queue.



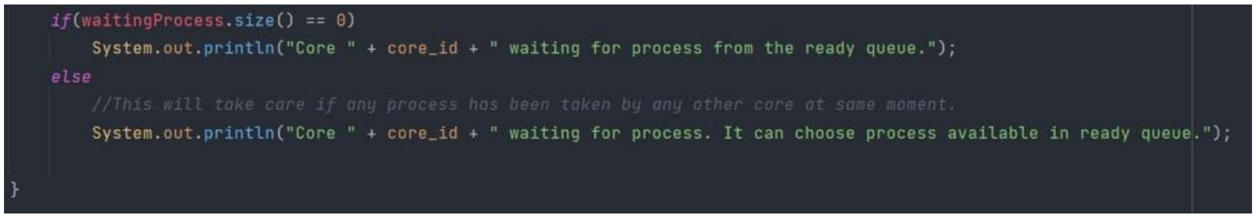
```java
public void cpuCoreAllocator(int core_id)
{
    this.core_id = core_id;
    Process process;

    synchronized (waitingProcess)
    {
        // Below logic check the number of process in the Queue.
        while(waitingProcess.size() == 0)
        {
            System.out.println("Core " + core_id + " is free.");
            if (Main.numOfFreeCores < Main.numOfCores)
                Main.numOfFreeCores ++;
            try
            {
                waitingProcess.wait();
            }
            catch(InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        System.out.println("Core "+ core_id + " is ready to use by a process from the queue.");
        process = (Process)((LinkedList<?>) waitingProcess).poll();
    }
```

Also in this part we can know which core is in use.



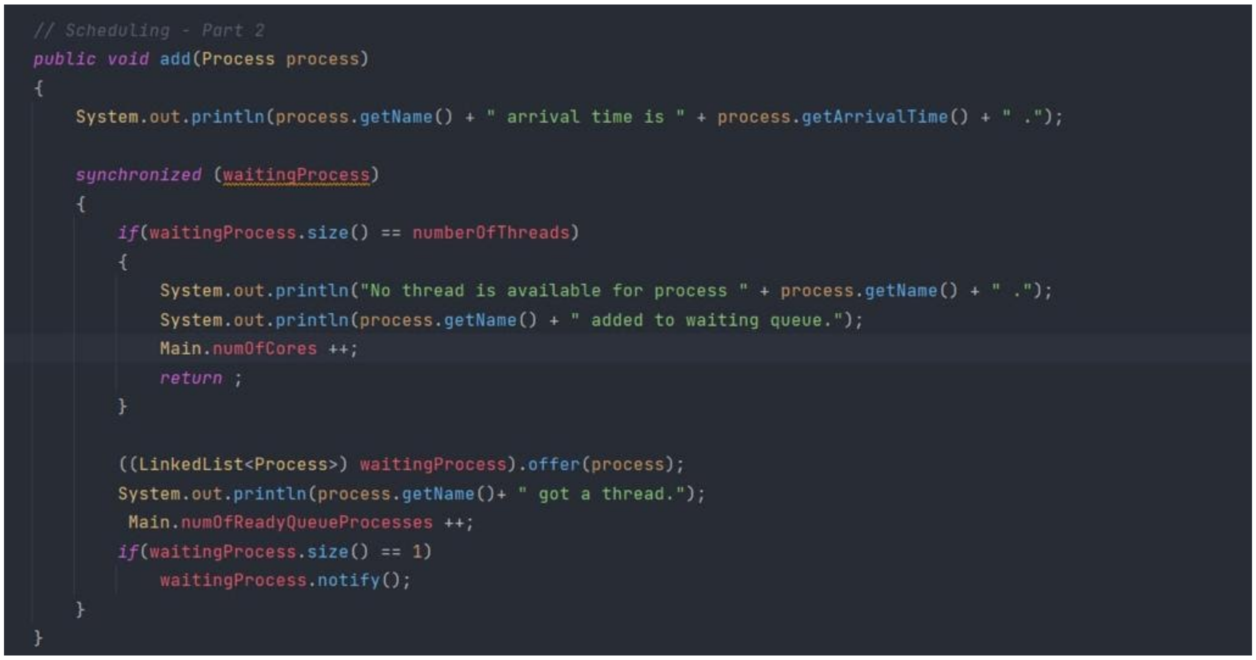
```
    long duration = 0;
    try
    {
        System.out.println(process.getName() + " is using core " + core_id + " .");
        Main.numOfRunningProcesses ++;
        if (Main.numOfReadyQueueProcesses > 0)
            Main.numOfReadyQueueProcesses --;
        if (Main.numOfCores > 0)
            Main.numOfFreeCores --;
        Main.numOfRunningProcesses ++;
        duration = (int)normal_distribution_generator.nextGaussian() * 5 + 15;    // Generating time for process burst time
        processes.add(process);
        TimeUnit.SECONDS.sleep(duration);
    }
    catch(InterruptedException e)
    {
        e.printStackTrace();
    }
    System.out.println(process.getName() + " release core " + core_id + " after its burst time " + duration+ " seconds.");
    process.setBurstTime((int) duration);
    process.coreNeeded --;
    Main.numOfRunningProcesses --;
    if(process.coreNeeded != 0) {
        add(process);
        Main.numOfReadyQueueProcesses ++;
    }
```

This section specifies that if a core is free to be processed, also, if a process is waiting in line, it can use it if the core is free.



```
    if(waitingProcess.size() == 0)
        System.out.println("Core " + core_id + " waiting for process from the ready queue.");
    else
        //This will take care if any process has been taken by any other core at same moment.
        System.out.println("Core " + core_id + " waiting for process. It can choose process available in ready queue.");

}
```

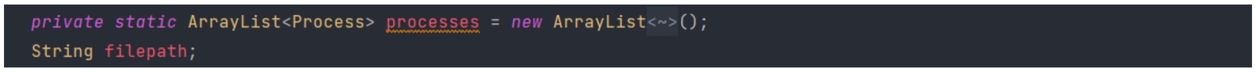
The last section is for the second part of scheduling that determines which process has to be in the waiting queue and when it has came and also print a message if the waiting process got a thread.



```java
// Scheduling - Part 2
public void add(Process process)
{
    System.out.println(process.getName() + " arrival time is " + process.getArrivalTime() + " .");

    synchronized (waitingProcess)
    {
        if(waitingProcess.size() == numberOfThreads)
        {
            System.out.println("No thread is available for process " + process.getName() + " .");
            System.out.println(process.getName() + " added to waiting queue.");
            Main.numOfCores ++;
            return ;
        }

        ((LinkedList<Process>) waitingProcess).offer(process);
        System.out.println(process.getName()+ " got a thread.");
         Main.numOfReadyQueueProcesses ++;
        if(waitingProcess.size() == 1)
            waitingProcess.notify();
    }
}
```

- **Scheduler**

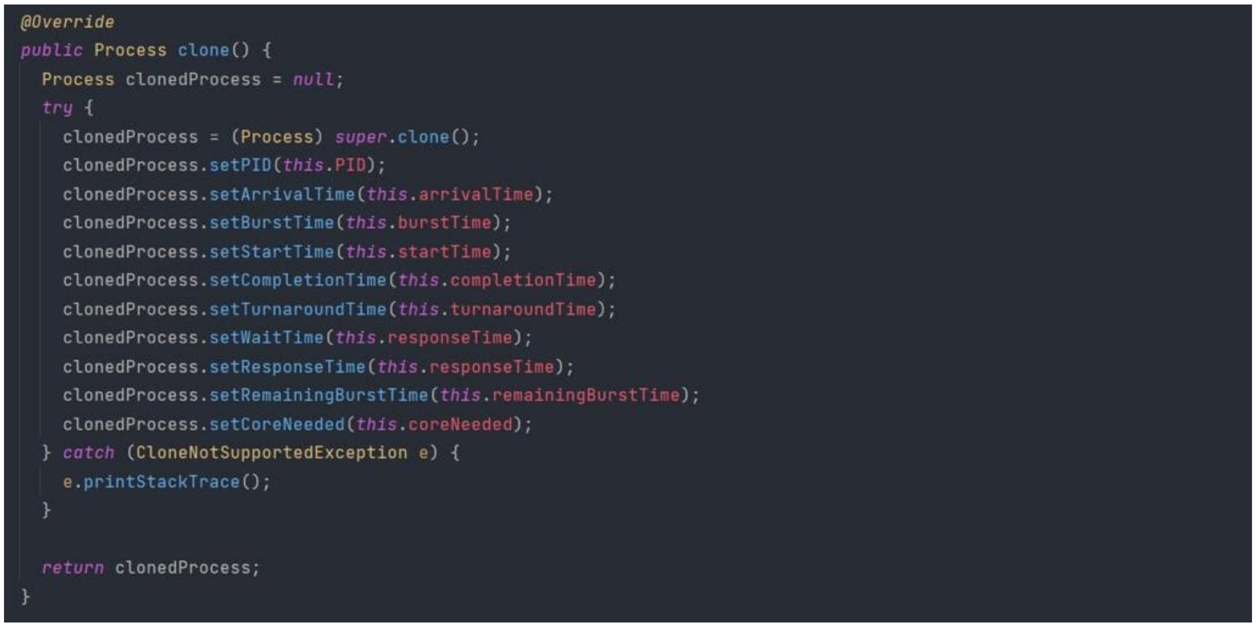
Parameters



```java
private static ArrayList<Process> processes = new ArrayList<~>();
String filepath;
```

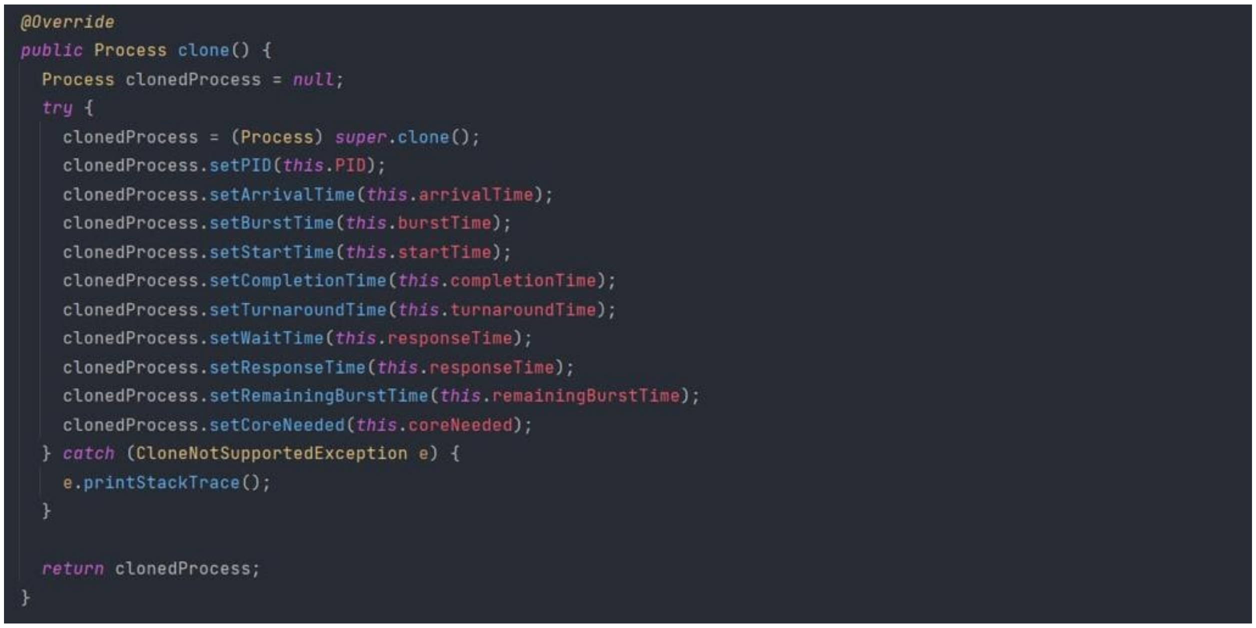
Process in scheduler package:

we are passing an ArrayList of Process objects to each scheduling algorithm, but when passing, object references are passed rather than object values so, we have to clone/deep copy the objects first.



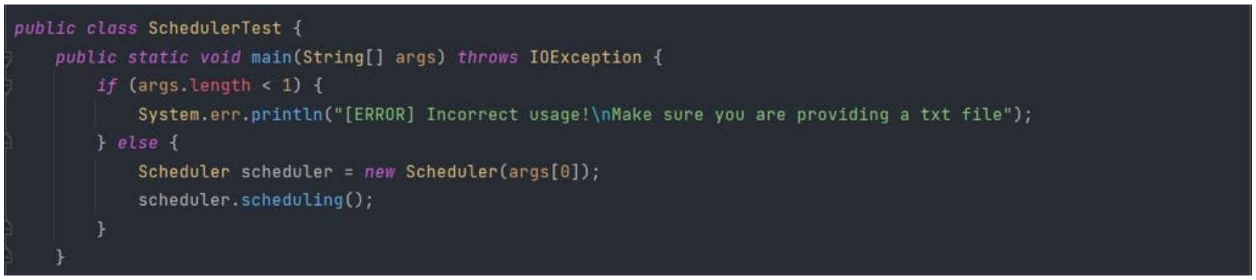
```java
@Override
public Process clone() {
  Process clonedProcess = null;
  try {
    clonedProcess = (Process) super.clone();
    clonedProcess.setPID(this.PID);
    clonedProcess.setArrivalTime(this.arrivalTime);
    clonedProcess.setBurstTime(this.burstTime);
    clonedProcess.setStartTime(this.startTime);
    clonedProcess.setCompletionTime(this.completionTime);
    clonedProcess.setTurnaroundTime(this.turnaroundTime);
    clonedProcess.setWaitTime(this.responseTime);
    clonedProcess.setResponseTime(this.responseTime);
    clonedProcess.setRemainingBurstTime(this.remainingBurstTime);
    clonedProcess.setCoreNeeded(this.coreNeeded);
  } catch (CloneNotSupportedException e) {
    e.printStackTrace();
  }

  return clonedProcess;
}
```

Each process is scheduled according to the number of cores required in the cores according to the same FCFS.



```java
@Override
public Process clone() {
  Process clonedProcess = null;
  try {
    clonedProcess = (Process) super.clone();
    clonedProcess.setPID(this.PID);
    clonedProcess.setArrivalTime(this.arrivalTime);
    clonedProcess.setBurstTime(this.burstTime);
    clonedProcess.setStartTime(this.startTime);
    clonedProcess.setCompletionTime(this.completionTime);
    clonedProcess.setTurnaroundTime(this.turnaroundTime);
    clonedProcess.setWaitTime(this.responseTime);
    clonedProcess.setResponseTime(this.responseTime);
    clonedProcess.setRemainingBurstTime(this.remainingBurstTime);
    clonedProcess.setCoreNeeded(this.coreNeeded);
  } catch (CloneNotSupportedException e) {
    e.printStackTrace();
  }

  return clonedProcess;
}
```

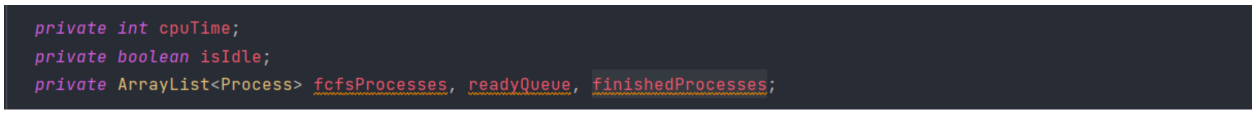
Scheduling Test



```
public class SchedulerTest {
    public static void main(String[] args) throws IOException {
        if (args.length < 1) {
            System.err.println("[ERROR] Incorrect usage!\nMake sure you are providing a txt file");
        } else {
            Scheduler scheduler = new Scheduler(args[0]);
            scheduler.scheduling();
        }
    }
```

- **FCFS**

Parameters



```
private int cpuTime;
private boolean isIdle;
private ArrayList<Process> fcfsProcesses, readyQueue, finishedProcesses;
```

This class expresses an implementation method of FCFS algorithm.

First Come First Serve (FCFS) is also known as First In First Out (FIFO) scheduling algorithm is the easiest and simplest CPU scheduling algorithm where the process which arrives first in the ready queue is executed first by the CPU. New process is executed only when the current process is executed fully by the CPU.

For implementing this algorithm we have to:

1- Input the processes along with their burst time.

2- Find waiting time for all processes.

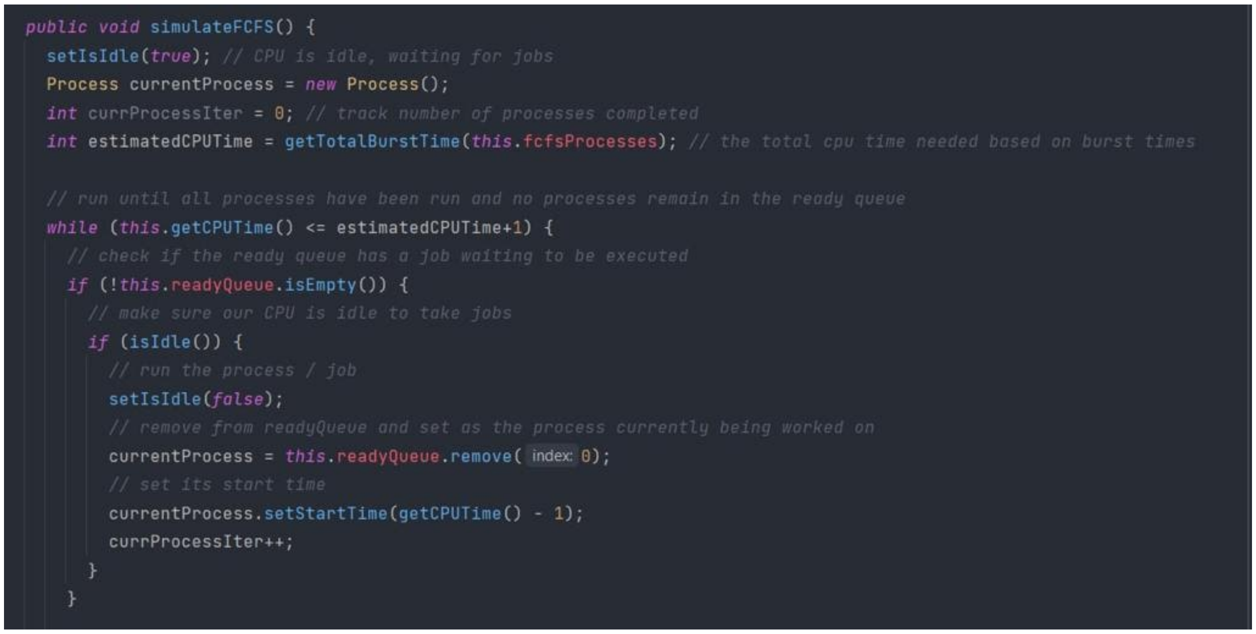
3- As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.

4- Find waiting time for all other processes i.e. for process i -> wt[i] = bt[i-1] + wt[i-1] .

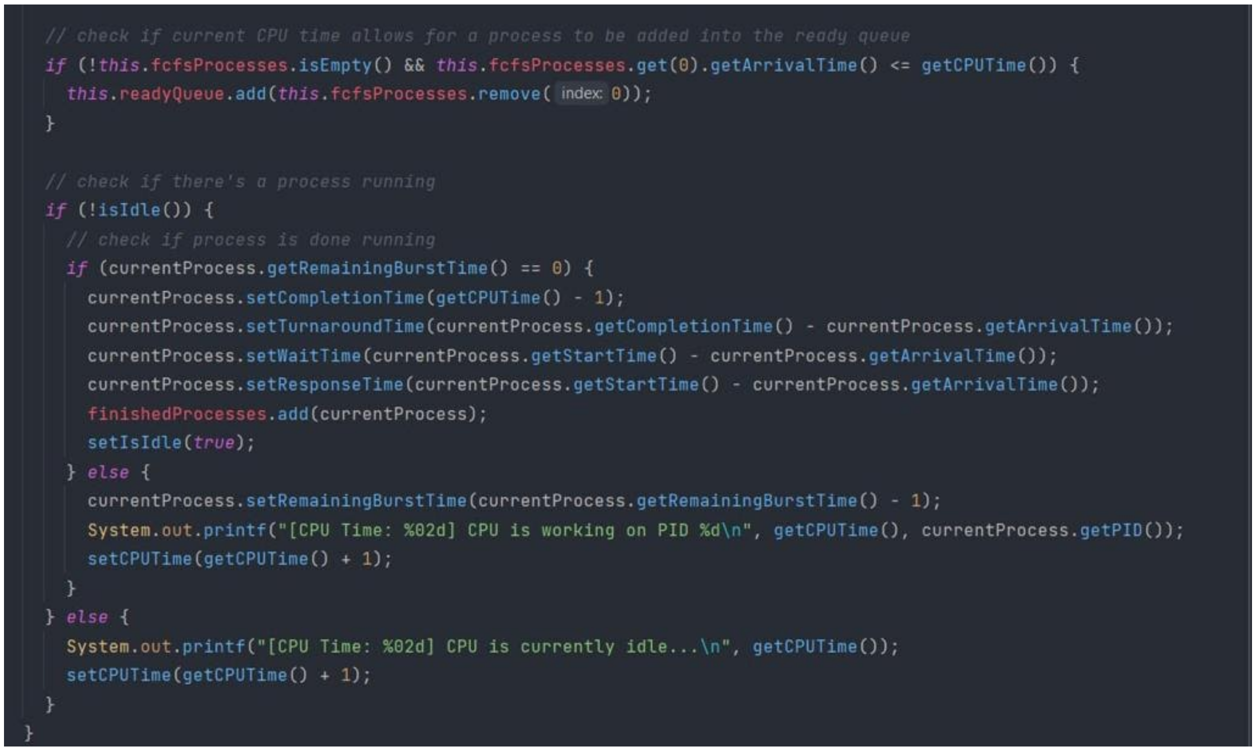
5- Find turnaround time = waiting_time + burst_time for all processes.

6- Find average waiting time = total_waiting_time / no_of_processes.

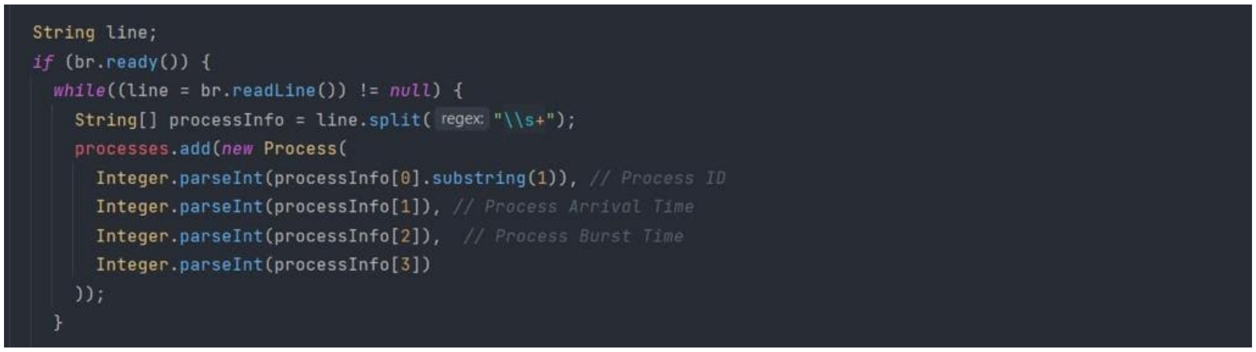
7- Similarly, find average turnaround time = total_turn_around_time / no_of_processes.



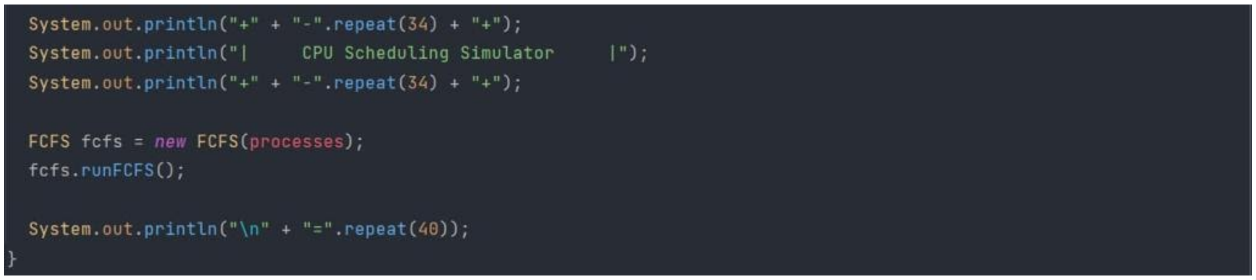
```
public void simulateFCFS() {
  setIsIdle(true); // CPU is idle, waiting for jobs
  Process currentProcess = new Process();
  int currProcessIter = 0; // track number of processes completed
  int estimatedCPUTime = getTotalBurstTime(this.fcfsProcesses); // the total cpu time needed based on burst times

  // run until all processes have been run and no processes remain in the ready queue
  while (this.getCPUTime() <= estimatedCPUTime+1) {
    // check if the ready queue has a job waiting to be executed
    if (!this.readyQueue.isEmpty()) {
      // make sure our CPU is idle to take jobs
      if (isIdle()) {
        // run the process / job
        setIsIdle(false);
        // remove from readyQueue and set as the process currently being worked on
        currentProcess = this.readyQueue.remove( index: 0);
        // set its start time
        currentProcess.setStartTime(getCPUTime() - 1);
        currProcessIter++;
      }
    }
```



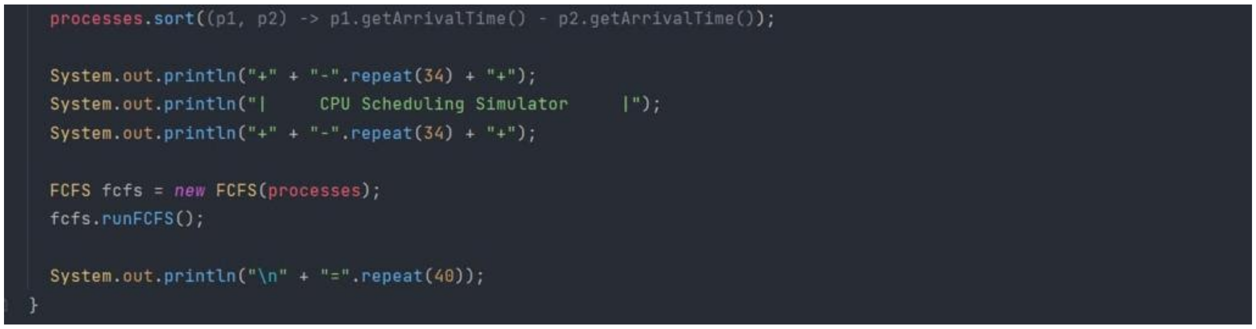
```
    // check if current CPU time allows for a process to be added into the ready queue
    if (!this.fcfsProcesses.isEmpty() && this.fcfsProcesses.get(0).getArrivalTime() <= getCPUTime()) {
      this.readyQueue.add(this.fcfsProcesses.remove( index: 0));
    }

    // check if there's a process running
    if (!isIdle()) {
      // check if process is done running
      if (currentProcess.getRemainingBurstTime() == 0) {
        currentProcess.setCompletionTime(getCPUTime() - 1);
        currentProcess.setTurnaroundTime(currentProcess.getCompletionTime() - currentProcess.getArrivalTime());
        currentProcess.setWaitTime(currentProcess.getStartTime() - currentProcess.getArrivalTime());
        currentProcess.setResponseTime(currentProcess.getStartTime() - currentProcess.getArrivalTime());
        finishedProcesses.add(currentProcess);
        setIsIdle(true);
      } else {
        currentProcess.setRemainingBurstTime(currentProcess.getRemainingBurstTime() - 1);
        System.out.printf("[CPU Time: %02d] CPU is working on PID %d\n", getCPUTime(), currentProcess.getPID());
        setCPUTime(getCPUTime() + 1);
      }
    } else {
      System.out.printf("[CPU Time: %02d] CPU is currently idle...\n", getCPUTime());
      setCPUTime(getCPUTime() + 1);
    }
  }
```



```
    String line;
    if (br.ready()) {
      while((line = br.readLine()) != null) {
        String[] processInfo = line.split( regex: "\\s+");
        processes.add(new Process(
          Integer.parseInt(processInfo[0].substring(1)), // Process ID
          Integer.parseInt(processInfo[1]), // Process Arrival Time
          Integer.parseInt(processInfo[2]),  // Process Burst Time
          Integer.parseInt(processInfo[3])
        ));
      }
```



```
  System.out.println("+" + "-".repeat(34) + "+");
  System.out.println("|     CPU Scheduling Simulator     |");
  System.out.println("+" + "-".repeat(34) + "+");

  FCFS fcfs = new FCFS(processes);
  fcfs.runFCFS();

  System.out.println("\n" + "=".repeat(40));
}
```

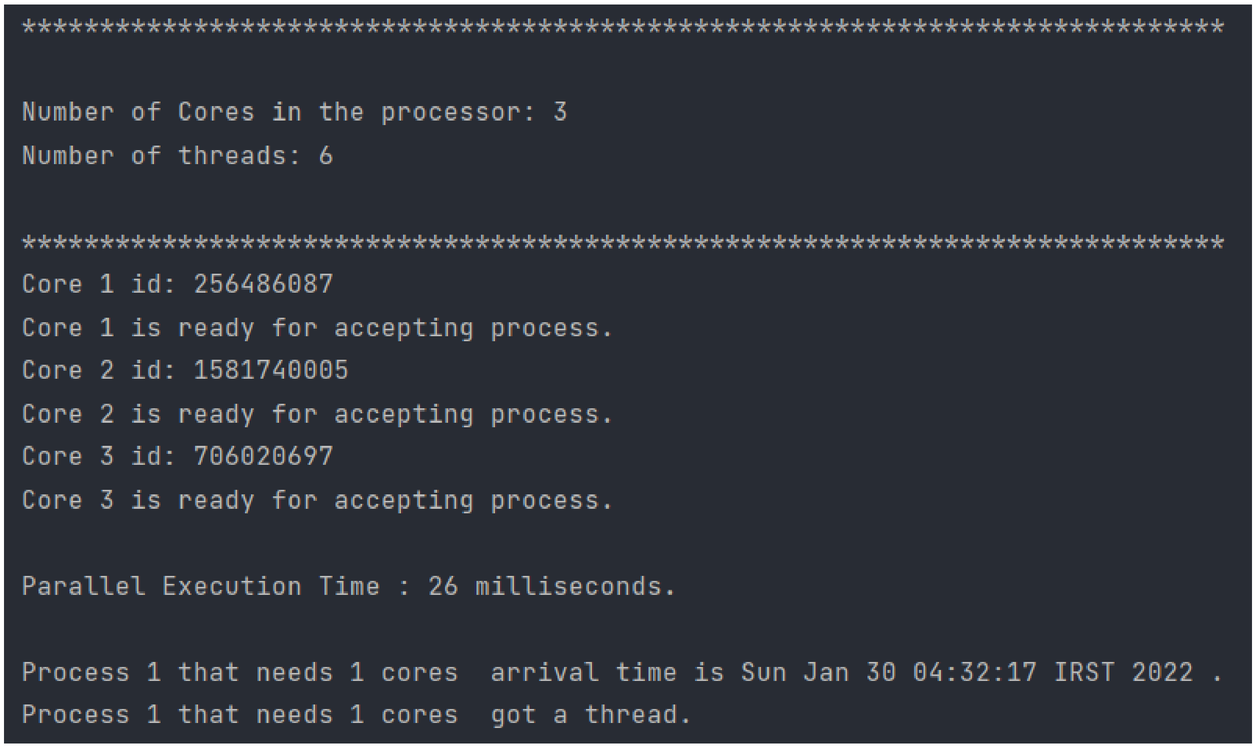


```
    processes.sort((p1, p2) -> p1.getArrivalTime() - p2.getArrivalTime());

    System.out.println("+" + "-".repeat(34) + "+");
    System.out.println("|     CPU Scheduling Simulator     |");
    System.out.println("+" + "-".repeat(34) + "+");

    FCFS fcfs = new FCFS(processes);
    fcfs.runFCFS();

    System.out.println("\n" + "=".repeat(40));
}
```

The above blocks are just used for reading our data and print results.

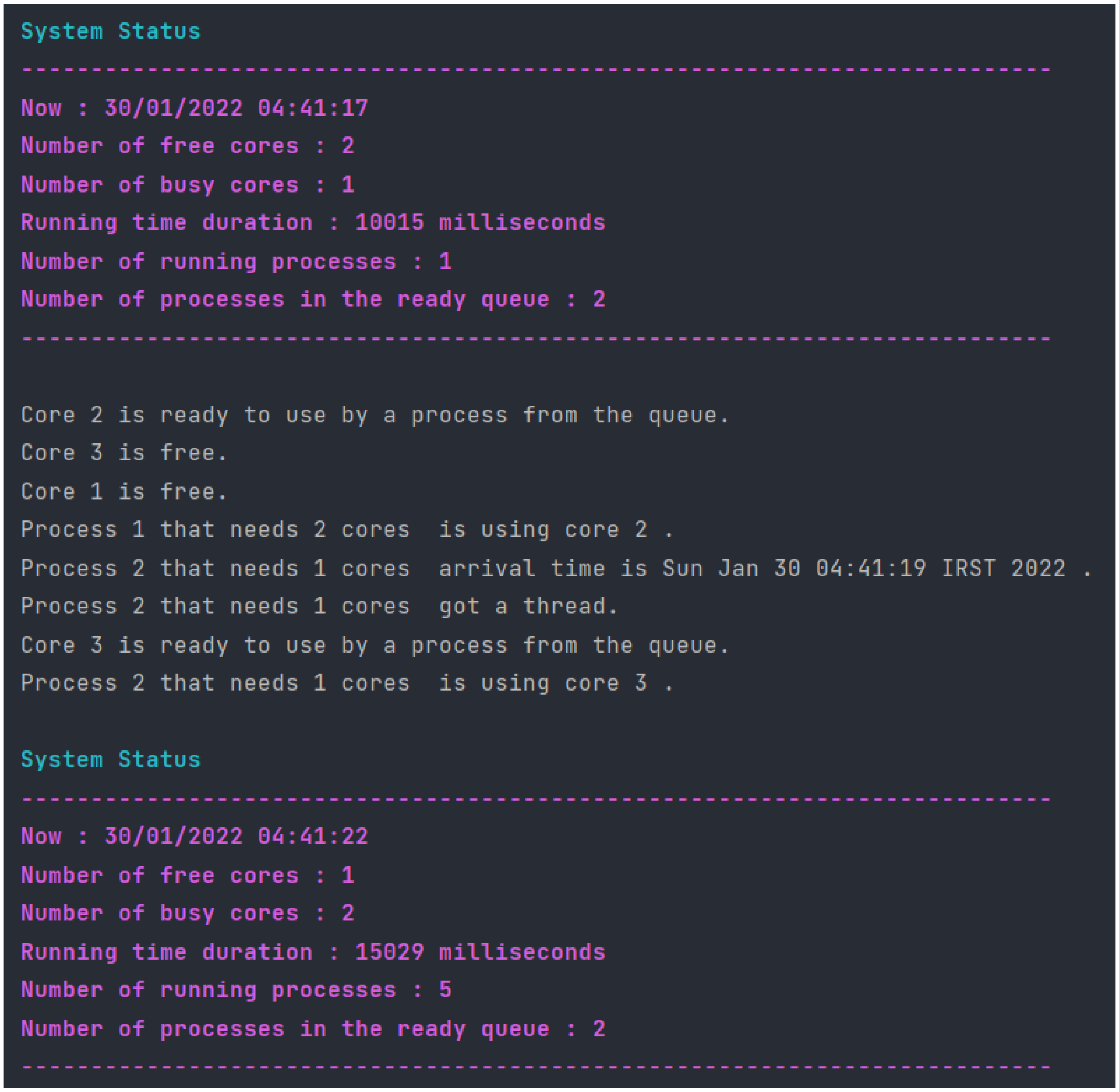
The Output of Main Program

Print details of cores and threads.



```
*****************************************************************************

Number of Cores in the processor: 3
Number of threads: 6

*****************************************************************************
Core 1 id: 256486087
Core 1 is ready for accepting process.
Core 2 id: 1581740005
Core 2 is ready for accepting process.
Core 3 id: 706020697
Core 3 is ready for accepting process.

Parallel Execution Time : 26 milliseconds.

Process 1 that needs 1 cores  arrival time is Sun Jan 30 04:32:17 IRST 2022 .
Process 1 that needs 1 cores  got a thread.
```

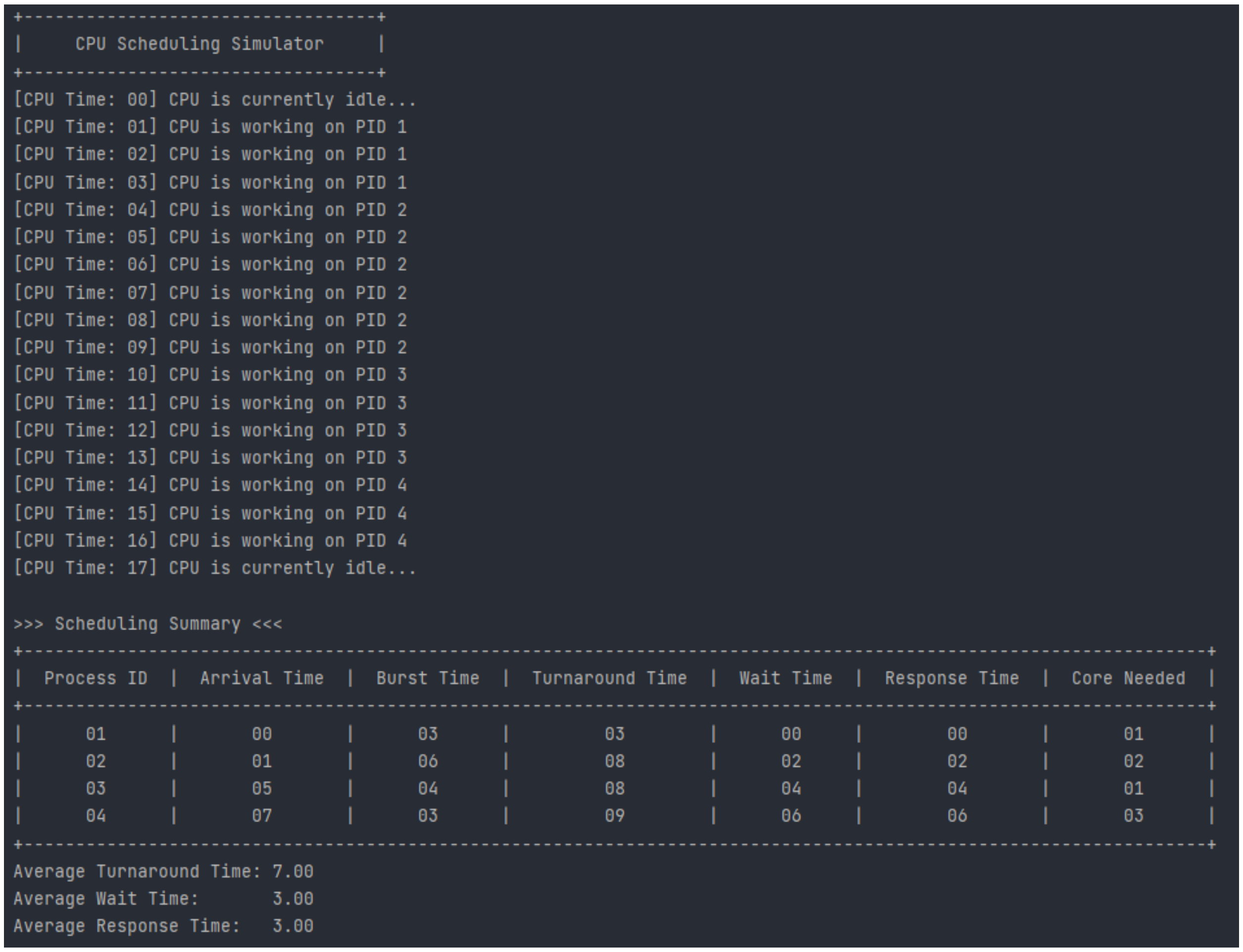
Print what happen at each time and also print system status in purple block in every 5 seconds.



```
System Status
---------------------------------------------------------------------------
Now : 30/01/2022 04:41:17
Number of free cores : 2
Number of busy cores : 1
Running time duration : 10015 milliseconds
Number of running processes : 1
Number of processes in the ready queue : 2
---------------------------------------------------------------------------


Core 2 is ready to use by a process from the queue.
Core 3 is free.
Core 1 is free.
Process 1 that needs 2 cores  is using core 2 .
Process 2 that needs 1 cores  arrival time is Sun Jan 30 04:41:19 IRST 2022 .
Process 2 that needs 1 cores  got a thread.
Core 3 is ready to use by a process from the queue.
Process 2 that needs 1 cores  is using core 3 .

System Status
---------------------------------------------------------------------------
Now : 30/01/2022 04:41:22
Number of free cores : 1
Number of busy cores : 2
Running time duration : 15029 milliseconds
Number of running processes : 5
Number of processes in the ready queue : 2
---------------------------------------------------------------------------
```

The Output of Scheduling Algorithm Separately

For testing the accuracy of our scheduling algorithm, we implemented it separately and test it by defined inputs.



```
+----------------------------------+
|     CPU Scheduling Simulator     |
+----------------------------------+
[CPU Time: 00] CPU is currently idle...
[CPU Time: 01] CPU is working on PID 1
[CPU Time: 02] CPU is working on PID 1
[CPU Time: 03] CPU is working on PID 1
[CPU Time: 04] CPU is working on PID 2
[CPU Time: 05] CPU is working on PID 2
[CPU Time: 06] CPU is working on PID 2
[CPU Time: 07] CPU is working on PID 2
[CPU Time: 08] CPU is working on PID 2
[CPU Time: 09] CPU is working on PID 2
[CPU Time: 10] CPU is working on PID 3
[CPU Time: 11] CPU is working on PID 3
[CPU Time: 12] CPU is working on PID 3
[CPU Time: 13] CPU is working on PID 3
[CPU Time: 14] CPU is working on PID 4
[CPU Time: 15] CPU is working on PID 4
[CPU Time: 16] CPU is working on PID 4
[CPU Time: 17] CPU is currently idle...

>>> Scheduling Summary <<<
+---------------------------------------------------------------------------------------------------------------------+
|  Process ID  |  Arrival Time  |  Burst Time  |  Turnaround Time  |  Wait Time  |  Response Time  |  Core Needed  |
+---------------------------------------------------------------------------------------------------------------------+
|      01      |       00       |      03      |        03         |     00      |       00        |      01       |
|      02      |       01       |      06      |        08         |     02      |       02        |      02       |
|      03      |       05       |      04      |        08         |     04      |       04        |      01       |
|      04      |       07       |      03      |        09         |     06      |       06        |      03       |
+---------------------------------------------------------------------------------------------------------------------+
Average Turnaround Time: 7.00
Average Wait Time:       3.00
Average Response Time:   3.00
```

How to run the program

Javac Main.java

Java Main 3 6

3 – number of cores

6 - number of threads

or passing the number of cores and number of threads as arguments to your IDE

Javac SchedueIrTest.java

Java SchedulerTest SchedulerTest1.txt

SchedulerTest1.txt - processes details

or passing .txt file that includes processes details as an argument in your IDE