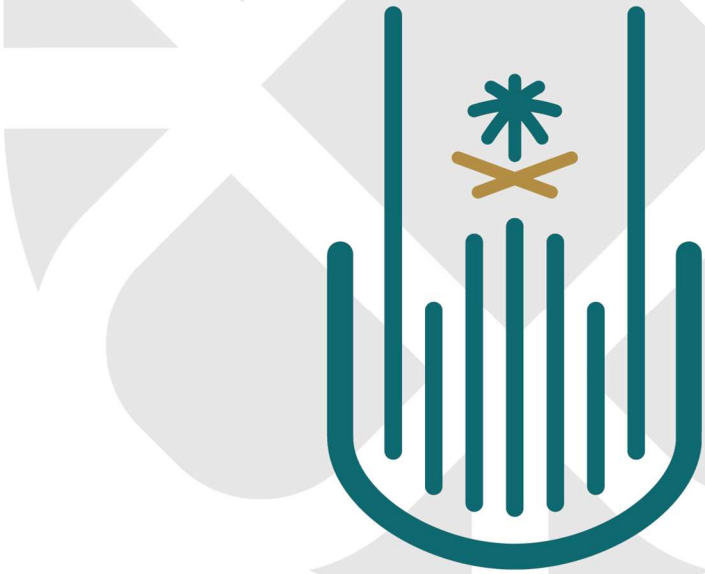

جامعة أم القرى
UMM AL-QURA UNIVERSITY

Operating System Scheduler

Students :

-Leader : Rayan Radin , 444004557
-Nadir Jam , 444000412
-Mohammed Bin Salman , 444001658
-Abdulbari Mulla , 444007134
-Abdulrahman Alkhamis , 444005843

Abstract:

In this project, we aim to implement different CPU scheduling algorithms We studied in this subject. We divided the project into files to make it simple and increase the readability of the project, we generally have 1 file named Test, it has all the 5 scheduling algorithms:

1- First-Come, First-Served (FCFS): The FCFS scheduling algorithm. It processes tasks in the order they arrive.

2- Shortest-Job-First (Non-Preemptive) (SJFNP): The SJFNP scheduling algorithm. It selects the job with the shortest execution time and runs it until it finishes.

3- Shortest-Job-First (Preemptive) (SJFP): The SJFP scheduling algorithm. It always picks the job with the shortest remaining time and can switch to a new job if it has a shorter remaining time.

4- Priority: The priority scheduling algorithm. It processes tasks based on their priority levels, with higher priority tasks being processed first.

5-Round Robin (RR): The Round Robin scheduling algorithm. It gives each task a fixed time slice before moving to the next task in a circular order.

Methodology:

1- First-Come, First-Served (FCFS):

Description: Takes a number of inputs and processes tasks in the order they arrive. This algorithm is simple and easy to implement, where first processes run first. (Made by: Nader Jam)

Methods and Functions:

1. Initialize arrays:

```
int[] burstTime = new int[processCount];
int[] waitingTime = new int[processCount];
int[] turnaroundTime = new int[processCount];
```

Description: initialize arrays to store burst time, waiting time and turnaround time for each process. The size of each array is stated by processCount.

2. Calculate waiting time and turnaround time:

```
waitingTime[0] = 0;
turnaroundTime[0] = burstTime[0];
for (int i = 1; i < processCount; i++) {
  waitingTime[i] = waitingTime[i - 1] + burstTime[i - 1];
  turnaroundTime[i] = waitingTime[i] + burstTime[i];
```

Description: Initialize the waiting time for the first process to 0 because it doesn't wait for any other process. Set the turnaround time for the first process equal to its burst time since it starts immediately. then Loop through each process and calculate Waiting Time and Turnaround Time

3. Output Details:

```
System.out.print("\n" + "process: ");
for (int i = 0; i < processCount; i++) {
  System.out.print("\t" + (i + 1));
}
```

Description: Print the process numbers in a row using for loop through each process. And do the same thing for Burst time, waiting time , and turnaround time.

4. Calculate and Print Average Waiting Time:

```
int sumWaitingTime = 0;
for (int i = 0; i < processCount; i++) {
    sumWaitingTime += waitingTime[i];
}
System.out.println("\n" + "Average Waiting Time: " + sumWaitingTime / (double) processCount);
```

Description: Calculate the total waiting time by summing up the waiting times of all processes. Compute the average waiting time by dividing the total waiting time by the number of processes and print it.

5. Calculate and Print Average Turnaround Time:

```
int sumTurnAroundTime = 0;
for (int i = 0; i < processCount; i++) {
    sumTurnAroundTime += turnaroundTime[i];
}
System.out.println("Average Turnaround Time: " + sumTurnAroundTime / (double) processCount);
```

Description: Calculate the total turnaround time by summing up the turnaround times of all processes. Compute the average turnaround time by dividing the total turnaround time by the number of processes and print it.

The output:

```
First-Come-First Served
How many processes do you want? 3
Enter burst time for process 1: 24
Enter burst time for process 2: 3
Enter burst time for process 3: 3

process:        1       2       3
Burst Time:     24      3       3
Waiting Time:   0       24      27
Turnaround Time: 24     27      30
Average Waiting Time: 17.0
Average Turnaround Time: 27.0
```

2- Shortest-Job-First (Non-Preemptive) (SJFNP):

Description: Selects the job with the shortest execution time and runs it until it finishes. (by: Abdulrahman Alkhamis)

Methods and Functions:

1. Initialize Arrays

```
int[] burstTimes = new int[processNumber];
int[] arrivalTimes = new int[processNumber];
int[] waitingTimes = new int[processNumber];
int[] turnaroundTimes = new int[processNumber];
boolean[] isCompleted = new boolean[processNumber];
```

Description: initialize arrays to store the burst time, arrival time, waiting time, turnaround time, and completion status for each process. The size of each array is determined by processNumber.

2. Input Burst and Arrival Times

```
for (int i = 0; i < processNumber; i++) {
  System.out.print("Enter burst time for process " + (i + 1) + ": ");
  burstTimes[i] = scnr.nextInt();
  System.out.print("Enter arrival time for process " + (i + 1) + ": ");
  arrivalTimes[i] = scnr.nextInt();
}
```

Description: This loop iterates over each process to get the burst time and arrival time from the user. The burst times and arrival times are stored in their respective arrays.

3. Initialization and Scheduling Loop:

```
int currentTime = 0;
int completed = 0;
while (completed != processNumber) {
```

Description: Initialize currentTime to 0, representing the current time of the scheduler. Initialize completed to 0, representing the number of processes that have been completed the loop continues until all processes are completed.

4. Finding Shortest Job:

```
if (shortestJobIndex == -1) {
    currentTime++;
} else {
    waitingTimes[shortestJobIndex] = currentTime - arrivalTimes[shortestJobIndex];
    currentTime += burstTimes[shortestJobIndex];
    turnaroundTimes[shortestJobIndex] = currentTime - arrivalTimes[shortestJobIndex];
    isCompleted[shortestJobIndex] = true;
    completed++;
```

Description: This loop iterates through all processes to find the shortest job (process) that has arrived, has not yet completed, and has the shortest burst time among eligible processes.

5. Handling Empty Time Slots:

```
int shortestJobIndex = -1;
int shortestBurst = Integer.MAX_VALUE;
```

Description: If no eligible job is found at the current time slot, increment currentTime to move to the next time slot.

6. Assigning Resources to Shortest Job:

```
for (int i = 0; i < processNumber; i++) {
    if (arrivalTimes[i] <= currentTime && !isCompleted[i] && burstTimes[i] < shortestBurst) {
        shortestBurst = burstTimes[i];
        shortestJobIndex = i;
```

Description: If a shortest job is found, assign resources to that job:
- Update waitingTimes for the selected job.
- Increment currentTime by the burst time of the selected job.
- Update turnaroundTimes for the selected job.
- Mark the selected job as completed in is Completed.
- Increment the count of completed processes.

The output:

```
Shortest-Job-First Non-Preemptive
How many processes do you want? 4
Enter burst time for process 1: 1
Enter arrival time for process 1: 0
Enter burst time for process 2: 4
Enter arrival time for process 2: 2
Enter burst time for process 3: 7
Enter arrival time for process 3: 3
Enter burst time for process 4: 5
Enter arrival time for process 4: 4

Process:        1       2       3       4
Arrival Time:   0       2       3       4
Burst Time:     1       4       7       5
Waiting Time:   0       0       8       2
Turnaround:     1       4       15      7
Average Waiting Time: 2.5
Average Turnaround Time: 6.75
```

3- Shortest-Job-First (Preemptive) (SJFP):

Description: picks the job with the shortest remaining time and can switch to a new job if it has a shorter remaining time.

Methods and Functions:

1. Initialize Arrays:

```
int number = scnr.nextInt();

int process[] = new int[number]; //initialize an array for the processes

int arrival[] = new int[number]; //initialize an array for the arrival time

int burst[] = new int[number]; //initialize an array for the burst time

int check[] = new int[number]; //initialize an array for the processes if it completed or not (flag)

int tempBurst[] = new int[number]; //initialize an array for the burst time after the update

int response[] = new int[number]; //initialize an array for the response time

Arrays.fill(response, -1); // make sure the array is not filled yet
```

Description: Initialize arrays to store processes, arrival times, burst times, temporary burst times (for calculation), and response times. The response array is filled with -1 initially to indicate that no process has yet received a response.

2. Input Process Details:

```
for (int i = 0; i < number; i++) {
  process[i] = i + 1;

  System.out.print("enter process " + (i + 1) + " arrival time: ");

  arrival[i] = scnr.nextInt();                                    // inputs

  System.out.print("enter process " + (i + 1) + " burst time: ");

  burst[i] = scnr.nextInt();

  tempBurst[i] = burst[i];

  check[i] = 0;
}
```

Description: Printing words asking the user to enter the arrival time and the burst time for each process.

3. Initialize Arrays:

```
int complete[] = new int[number]; //initialize an array for the completion time for the processes

int timer = 0; //initialize an array for timer (Clock)

int totalProcces = 0; //initialize an array for completed processes
```

Description: Initialize arrays to store the completion time for processes, timer for the clock timer, total process for the completed process.

4. SJF Preemptive scheduling:

```
if (totalProcces == number)
  break;

for (int i = 0; i < number; i++) {
  if ((arrival[i] <= timer) && (check[i] == 0) && (burst[i] < min)) {
    min = burst[i];
    current = i;
  }
}
```

Description: first the code will check if all processes have been completed. If the total number of completed processes equals the total number of processes, the loop breaks.

If it didn't break, the loop will iterate over all processes to find the process with the shortest remaining burst time that has already arrived and has not yet been completed.

```
if (current == number)
  timer++;
```

Description: Check If current is still equal to number, if it is, it means no process was found that is ready to execute at the current time.

```
else {

  if (response[current] == -1) {
    response[current] = timer - arrival[current];
  }

  burst[current]--;
  timer++;

  if (burst[current] == 0) {
    complete[current] = timer;

    check[current] = 1;

    totalProcces++;
  }
}
```

Description: If this is the first time the current process is being executed, the response time is recorded as the difference between the current timer value and the arrival time of the process.

The burst time of the current process is decreased by 1, and the timer will be increased to go to the next second.

If the burst time of the current process reaches zero, it means the process has completed its execution, the completion time will be recorded in complete array, and the total number of completed processes will be increased.

5. Initialize Arrays:

```
int turnAround[] = new int[number]; //initialize an array for turnaround time

int waiting[] = new int[number]; //initialize an array for waiting time
```

Description: Initialize arrays to store turnaround time, and waiting time

6. Initialize the average:

```
double averageWating = 0;

double averageTurnAround = 0;            // initialize the average

double averageResponse = 0;
```

Description: Initialize variables to store the average for the waiting time, turnaround time, and response time.

7. Calculate the average:

```
for (int i = 0; i < number; i++) {
  turnAround[i] = complete[i] - arrival[i];

  waiting[i] = turnAround[i] - tempBurst[i];

  averageWating += waiting[i];                //calculate the average

  averageTurnAround += turnAround[i];

  averageResponse += response[i];

}
```

Description: make a for loop to calculate and store and the average.

8. Print the result

```
System.out.print("response = " + "\t");
for (int i = 0; i < number; i++) {
  System.out.print(response[i] + "\t");
}
System.out.println();


System.out.println("\naverage turnaround time is: " + (double) (averageTurnAround / number));
```

Description: last thing the code will print every array content, than it going to print the average.

4- Priority:

Description: It processes tasks based on their priority levels, with higher priority tasks being processed first.

Methods and Functions:

1. Initialize Arrays:

```
int[] priorities = new int[processCounts];
int[] bursts = new int[processCounts];
int[] waitingtimes = new int[processCounts];
int[] turnaround = new int[processCounts];
int[] processIds = new int[processCounts];
```

Description: initialize arrays to store priorities, burst time, waiting time, turnaround time and processIds for each process. The size of each array is stated by processCount.

2. Input Process Details:

```
for (int i = 0; i < processCounts; i++) {
  processIds[i] = i + 1;
  System.out.print("Enter priority for process " + processIds[i] + ": ");
  priorities[i] = scnr.nextInt();

  System.out.print("Enter burst time for process " + processIds[i] + ": ");
  bursts[i] = scnr.nextInt();
```

Description: Prompt the user to enter priority and burst time for each process, and store the values in their respective arrays.

```
for (int i = 0; i < processCounts - 1; i++) {
  for (int j = 0; j < processCounts - i - 1; j++) {
    if (priorities[j] > priorities[j + 1]) {
      // Swap priorities
      int tempPriority = priorities[j];
      priorities[j] = priorities[j + 1];
      priorities[j + 1] = tempPriority;
```

Description: Sort the processes based on their priority using the Bubble Sort algorithm. Swaps the priorities, burst times, and process IDs of the processes. This ensures that the processes are rearranged according to their priorities in ascending order.

4. Calculate waiting time and turnaround time:

```
waitingtimes[0] = 0; // Waiting time for the first process is always 0
for (int i = 1; i < processCounts; i++) {
  waitingtimes[i] = waitingtimes[i - 1] + bursts[i - 1];
}

// Calculate turnaround times
for (int i = 0; i < processCounts; i++) {
  turnaround[i] = waitingtimes[i] + bursts[i];
}
```

Description: Initialize the waiting time for the first process to 0 because it doesn't wait for any other process. Set the turnaround time for the first process equal to its burst time since it starts immediately. then Loop through each process and calculate Waiting Time and Turnaround Time.

5. Output Details:

```
System.out.print("\n" + "Priority: ");
for (int i = 0; i < processCounts; i++) {   //first, print Priority
  System.out.print("\t" + priorities[i]);
```

Description: Print the priority in a row using for loop through each process. And do the same thing for process, burst time, waiting time, and turnaround time.

6. The output:

```
Priority
Enter the number of processes: 5
Enter priority for process 1: 3
Enter burst time for process 1: 10
Enter priority for process 2: 1
Enter burst time for process 2: 1
Enter priority for process 3: 4
Enter burst time for process 3: 2
Enter priority for process 4: 5
Enter burst time for process 4: 1
Enter priority for process 5: 2
Enter burst time for process 5: 5

Priority:       1       2       3       4       5
Process:        2       5       1       3       4
Burst Time:     1       5       10      2       1
Waiting Time:   0       1       6       16      18
Turnaround:     1       6       16      18      19
Average Waiting Time: 8.2
Average Turnaround Time: 12.0
```

5- Round Robin (RR):

Description: It gives each task a fixed time slice before moving to the next task in a circular order.

Methods and Functions:

1. Initialize Arrays for Burst Times and Arrival Times:

```
private int timeQuantum;
private int[] burstTime;
private int[] arrivalTime;
private int[] waitingTime;
private int[] turnaroundTime;
private int[] responseTime;
private ArrayList<Integer> executionOrder;
```

Description: These variables hold information related to the Round Robin scheduling algorithm, such as time quantum, burst times, arrival times, waiting times, turnaround times, response times, and execution order.

2. Constructor:

```
public RR(int timeQuantum, int[] burstTime, int[] arrivalTime) {
    this.timeQuantum = timeQuantum;
    this.burstTime = burstTime;
    this.arrivalTime = arrivalTime;
    int numOfProcesses = burstTime.length;  // it's equal to number of processes
    this.waitingTime = new int[numOfProcesses];
    this.turnaroundTime = new int[numOfProcesses];
    this.responseTime = new int[numOfProcesses];
    this.executionOrder = new ArrayList<>();
    applyAlgorithm();
}
```

Description: Initializes an instance of the Round Robin scheduler with the provided time quantum, burst times, and arrival times.
Calculates waiting times, turnaround times, response times, and the execution order by calling the applyAlgorithm() method.

3. process Initialize:

```
private void applyAlgorithm() {
    int numOfProcesses = burstTime.length;
    int[] remainingBurstTime = new int[numOfProcesses];
    boolean[] isFirstResponse = new boolean[numOfProcesses];
    for (int i = 0; i < numOfProcesses; i++) {
        remainingBurstTime[i] = burstTime[i];
        isFirstResponse[i] = true;
    }
```

Description: Iterates over each process to initialize the remaining burst time and set the first response flag to true.

4. First Response Check:

```
while (!queue.isEmpty()) {
    int processIndex = queue.poll();
    if (isFirstResponse[processIndex]) {
        responseTime[processIndex] = currentTime - arrivalTime[processIndex];
        isFirstResponse[processIndex] = false;
    }
```

Description: Enters a loop that continues until the queue is empty. Retrieves a process index from the queue, initializes variables for current time and a queue to store the indexes of processes to be executed. Adds the first process to the queue and marks it as in the queue.

5. Time Quantum Execution:

```
    if (remainingBurstTime[processIndex] > timeQuantum) {
        currentTime += timeQuantum;
        remainingBurstTime[processIndex] -= timeQuantum;
        executionOrder.add(processIndex + 1);
    } else {
        currentTime += remainingBurstTime[processIndex];
        waitingTime[processIndex] = currentTime - burstTime[processIndex] - arrivalTime[processIndex];
        remainingBurstTime[processIndex] = 0;
        executionOrder.add(processIndex + 1);
    }
```

Description: Checks if the remaining burst time of the process is greater than the time quantum: If true, executes the process for the time quantum, updates the remaining burst time, and adds the process index to the execution order. If false, executes the process until completion, calculates its waiting time, updates the remaining burst time to 0, and adds the process index to the execution order.

6. Arrival Time Check and Queue Management:

```
    for (int i = 1; i < numOfProcesses; i++) {
        if (arrivalTime[i] <= currentTime && remainingBurstTime[i] > 0 && !isInQueue[i]) {
            queue.add(i);
            isInQueue[i] = true;
        }
    }
    if (remainingBurstTime[processIndex] > 0) {
        queue.add(processIndex);
    }
}
```

Description: Iterates over the remaining processes to check if any new processes have arrived while others were executing.
If a new process has arrived and meets certain conditions (arrival time <= current time, remaining burst time > 0, not already in the queue), it's added to the queue, If the current process still has remaining burst time after execution, it's added back to the queue for further execution.

7.Turnaround Time Calculation:

```
for (int i = 0; i < numOfProcesses; i++) {
    turnaroundTime[i] = burstTime[i] + waitingTime[i];
}
```

Description: After the loop completes, calculates the turnaround time for each process by adding its burst time and waiting time.

8. Process Details:

```
System.out.print("\n" + "Process:\t");
for (int i = 0; i < burstTime.length; i++) {
    System.out.print( "\tP" + (i + 1));
}
```

Description: Print the process numbers in a row using for loop through each process. And do the same thing for Burst time, waiting time, Arrival time, waiting time, turnaround time and response time.

9. Average Times:

```
double sumWaitingTimes = 0;
  for (int i = 0; i < burstTime.length; i++) {
    sumWaitingTimes += waitingTime[i];
  }

  double sumTurnAroundTime = 0;
  for (int i = 0; i < burstTime.length; i++) {
    sumTurnAroundTime += turnaroundTime[i];
  }

  double sumResponseTime = 0;
  for (int i = 0; i < burstTime.length; i++) {
    sumResponseTime += responseTime[i];
  }
```

Description: Calculates the average waiting time, average turnaround time and average response time.

10. print Average Times and Execution Order:

```
System.out.printf("%n%naverage waiting time is: %.3f%n", (sumWaitingTimes / burstTime.length));
System.out.printf("average turnaround time is: %.3f%n", (sumTurnAroundTime / burstTime.length));
System.out.printf("average response time is: %.3f%n", (sumResponseTime / burstTime.length));

System.out.println("\nExecution Order: " + executionOrder);
```

Description: print Average Times and various statistics related to the process scheduling and execution.

11. Output:

```
How long time quantum (ms)? 4
How many processes do you want? 3

Are all the processes arrive at the same time 0? (if No, you will enter arrival time for each one)
               1.Yes   2.No
1
Enter burst time for process 1: 24
Enter burst time for process 2: 3
Enter burst time for process 3: 3

Process:               P1      P2      P3
Burst Time:            24      3       3
Arrival Time:          0       0       0
Waiting Time:          6       4       7
Turnaround Time:       30      7       10
Response Time:         0       4       7

average waiting time is: 5.667
average turnaround time is: 15.667
average response time is: 3.667

Execution Order: [1, 2, 3, 1, 1, 1, 1, 1]
```

Challenges We Faced:

- Connecting the lectures and the programming language.
- Syncing between the arrays.
- Finding information in programing language.
- The goal of the project wasn't clear in the beginning of the semester.
- Couldn't start the project early because we were still studying the subject.
- Hard to combine members codes.

- **Can't find information.**
- **The goal of the project wasn't clear in the beginning.**
- **Couldn't start project early because we were still studying the subject.**
- **It was hard to combine all the codes together.**
- **Not enough programming information.**

Conclusion: In this project, we implemented and compared different CPU scheduling algorithms to see how they manage tasks and affect performance.
Here's a summary of what we learned:

1. First-Come, First-Served (FCFS
   - Simple to understand and implement.
   - Can cause delays for short tasks if long tasks come first.

2. Shortest-Job-First (Non-Preemptive) (SJFNP
   - Reduces average waiting time by always running the shortest task next.

- Long tasks can wait a long time if many short tasks keep coming.

3. Shortest-Job-First (Preemptive) (SJFP
   - Improves response time by running the task with the shortest remaining time.
   - More complex because it constantly checks for the shortest remaining task.

4. Priority Scheduling
   - Runs tasks based on priority levels.
   - Can cause low-priority tasks to wait a long time.

5. Round Robin (RR
   - Fairly shares CPU time among tasks in a circular order.
   - Needs a well-chosen time slice to work effectively.

By breaking the project into separate files for each algorithm, we made the code easier to manage and understand. Each algorithm has its pros and cons, making them suitable for different situations.

Overall, this project helped us see how different scheduling algorithms work and their impact on system performance. Understanding these algorithms helps in choosing the right one for specific needs, improving efficiency and responsiveness.

References:

https://www.w3schools.com/
https://www.geeksforgeeks.org/
https://stackoverflow.com/
https://chatgpt.com/

Our programing site:

https://github.com/rayanAdeb/Project