

ChatGPTChatGPT



```
import java.util.*;

class Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int priority;
    int completionTime;
    int turnaroundTime;
    int waitingTime;
}

public class PriorityScheduling {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of processes: ");
        int n = scanner.nextInt();

        Process[] processes = new Process[n];

        for (int i = 0; i < n; i++) {
            processes[i] = new Process();
            System.out.print("Enter arrival time for process " + (i + 1) + ": ");
            processes[i].arrivalTime = scanner.nextInt();
            System.out.print("Enter burst time for process " + (i + 1) + ": ");
            processes[i].burstTime = scanner.nextInt();
            System.out.print("Enter priority for process " + (i + 1) + ": ");
            processes[i].priority = scanner.nextInt();
            processes[i].pid = i + 1;
        }

        Arrays.sort(processes, Comparator.comparingInt(p -> p.arrivalTime));

        int currentTime = 0;
        boolean[] processed = new boolean[n];
        int completedProcesses = 0;

        while (completedProcesses < n) {
            int idx = -1;
            int minPriority = Integer.MAX_VALUE;

            for (int i = 0; i < n; i++) {
                if (!processed[i] && processes[i].arrivalTime <= currentTime && processes[i].priority <

```

```

minPriority) {
    minPriority = processes[i].priority;
    idx = i;
}
}

if (idx != -1) {
    processed[idx] = true;
    processes[idx].completionTime = currentTime + processes[idx].burstTime;
    processes[idx].turnaroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
    processes[idx].waitingTime = processes[idx].turnaroundTime - processes[idx].burstTime;
    currentTime = processes[idx].completionTime;
    completedProcesses++;
} else {
    currentTime++;
}
}

System.out.println("\nPID\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround
Time\tWaiting Time");
for (int i = 0; i < n; i++) {
    System.out.println(processes[i].pid + "\t" + processes[i].arrivalTime + "\t\t" +
processes[i].burstTime + "\t\t" + processes[i].priority + "\t\t" + processes[i].completionTime + "\t\t" +
processes[i].turnaroundTime + "\t\t" + processes[i].waitingTime);
}

scanner.close();
}
}

```



Code Explanation:

The provided Java code implements a **Priority Scheduling** algorithm for process management in an operating system. This algorithm schedules processes based on their priorities, executing the process with the highest priority (smallest numerical value) first, among those that have arrived.

Step-by-Step Description:

1. Process Class Definition:

- Each process is represented as an object of the ``Process`` class.
- Contains attributes:
 - ``pid`` (Process ID)
 - ``arrivalTime``
 - ``burstTime`` (execution time)
 - ``priority`` (lower value = higher priority)
 - ``completionTime``

- ``turnaroundTime`` (TAT)
- ``waitingTime``

2. User Input:

- The user specifies the number of processes.
- For each process, the user provides:
 - Arrival time
 - Burst time
 - Priority

3. Sorting by Arrival Time:

- The processes are sorted in ascending order based on ``arrivalTime`` to ensure they are considered in the correct sequence.

4. Scheduling Logic:

- A loop iterates until all processes are scheduled.
- At each iteration:
 - Among processes that have arrived and are not yet processed, the one with the **highest priority** (lowest numerical value) is selected.
 - If no process is ready (i.e., none have arrived at ``currentTime``), the ``currentTime`` is incremented.
- Once a process is selected:
 - Its ``completionTime`` is calculated (``currentTime + burstTime``).
 - ``turnaroundTime = completionTime - arrivalTime``
 - ``waitingTime = turnaroundTime - burstTime``
 - The ``currentTime`` is updated to reflect the process's completion.

5. Output:

- After all processes are scheduled, the program prints a table showing:
 - Process ID
 - Arrival time
 - Burst time
 - Priority
 - Completion time
 - Turnaround time
 - Waiting time

Example Input and Output:

Input:

```
mathematica
```

```
Enter the number of processes: 3
Enter arrival time for process 1: 0
Enter burst time for process 1: 10
Enter priority for process 1: 2
Enter arrival time for process 2: 1
Enter burst time for process 2: 5
Enter priority for process 2: 1
Enter arrival time for process 3: 2
Enter burst time for process 3: 8
Enter priority for process 3: 3
```

Output:

CSS

PID	Arrival Time	Burst Time	Priority	Completion Time	Turnaround Time	Wait
2	1	5	1	6	5	0
1	0	10	2	16	16	6
3	2	8	3	24	22	14

Key Features:

- **Non-Preemptive:** Once a process starts execution, it runs to completion.
- **Idle Time Handling:** If no process has arrived, the scheduler waits by incrementing ``currentTime``.
- **Priority Consideration:** Only processes that have arrived and are not yet completed are considered for execution.

Potential Enhancements:

- Add support for **preemptive priority scheduling**.
- Include error handling for invalid inputs.
- Allow ties in priorities to be broken by arrival time or process ID.



in detail



Detailed Explanation of the Priority Scheduling Code

This Java program implements **non-preemptive priority scheduling**, where processes are selected based on their priority, and once selected, they run to completion. Let's break the program into logical components for a thorough understanding:

1. Process Class

The ``Process`` class represents each process with the following fields:

Field	Description
<code>`pid`</code>	Unique process ID (set sequentially starting from 1).
<code>`arrivalTime`</code>	The time at which the process arrives in the ready queue.
<code>`burstTime`</code>	The amount of CPU time required to complete the process.
<code>`priority`</code>	Priority of the process (lower value indicates higher priority).
<code>`completionTime`</code>	The time at which the process finishes execution.
<code>`turnaroundTime`</code>	The total time taken from arrival to completion (<code>`completionTime` - <code>`arrivalTime`</code></code>).
<code>`waitingTime`</code>	The total time spent waiting in the ready queue (<code>`turnaroundTime` - <code>`burstTime`</code></code>).

2. Input and Initialization

```
java
```

```
System.out.print("Enter the number of processes: ");
int n = scanner.nextInt();
```

- The user specifies the number of processes (`n`).

For each process:

```
java
```

```
processes[i] = new Process();
System.out.print("Enter arrival time for process " + (i + 1) + ": ");
processes[i].arrivalTime = scanner.nextInt();
System.out.print("Enter burst time for process " + (i + 1) + ": ");
processes[i].burstTime = scanner.nextInt();
System.out.print("Enter priority for process " + (i + 1) + ": ");
processes[i].priority = scanner.nextInt();
processes[i].pid = i + 1;
```

- The program collects **arrival time**, **burst time**, and **priority** for each process.
- The `pid` field is initialized sequentially (`i + 1`) to uniquely identify each process.

3. Sorting by Arrival Time

```
java
```

```
Arrays.sort(processes, Comparator.comparingInt(p -> p.arrivalTime));
```

The processes are sorted in ascending order of `arrivalTime`. This ensures that processes are considered for scheduling in the order they arrive. Sorting is necessary for correctly handling arrival times and idle CPU periods.

4. Scheduling Logic

The main logic of the program uses a loop to process all `n` processes:

Key Variables:

Variable	Description
`currentTime`	Tracks the current time in the system.
`processed`	A boolean array indicating whether a process is already completed.
`completedProcesses`	Counts the number of processes that have been completed.
`idx`	Holds the index of the next process to execute based on the priority check.

Variable	Description
<code>`minPriority`</code>	Tracks the smallest priority value among eligible processes.

Loop Execution:

```
java

while (completedProcesses < n) {
    int idx = -1;
    int minPriority = Integer.MAX_VALUE;
```

- The loop runs until all processes are completed (``completedProcesses < n``).
- Variables ``idx`` and ``minPriority`` are used to find the next eligible process with the highest priority (lowest numerical value).

Process Selection:

```
java

for (int i = 0; i < n; i++) {
    if (!processed[i] && processes[i].arrivalTime <= currentTime && processes[i].priority <
minPriority) {
        minPriority = processes[i].priority;
        idx = i;
    }
}
```

- The code iterates over all processes to check:
 1. The process has not been completed (``!processed[i]``).
 2. The process has arrived (``processes[i].arrivalTime <= currentTime``).
 3. The process has the smallest priority (``processes[i].priority < minPriority``).
- If no eligible process is found (``idx == -1``), the CPU waits for the next process to arrive:

```
java

if (idx != -1) {
    // Process the selected process
} else {
    currentTime++;
}
```

Process Execution:

When a process is selected (``idx != -1``):

```
java

processed[idx] = true;
processes[idx].completionTime = currentTime + processes[idx].burstTime;
```

```
processes[idx].turnaroundTime = processes[idx].completionTime - processes[idx].arrivalTime;
processes[idx].waitingTime = processes[idx].turnaroundTime - processes[idx].burstTime;
currentTime = processes[idx].completionTime;
completedProcesses++;
```

- **Mark Process as Completed:** `processed[idx] = true`.`
- **Calculate Completion Time:** `currentTime + burstTime`.`
- **Calculate Turnaround Time (TAT):** `completionTime - arrivalTime`.`
- **Calculate Waiting Time:** `turnaroundTime - burstTime`.`
- **Update Current Time:** The `currentTime`` is updated to reflect the completion of the selected process.
- **Increment Completed Processes Counter:** `completedProcesses++`.`

5. Output Results

Finally, the program prints the results in a tabular format:

```
java
```

```
System.out.println("\nPID\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround T
ime\tWaiting Time");
for (int i = 0; i < n; i++) {
    System.out.println(processes[i].pid + "\t" + processes[i].arrivalTime + "\t\t" + process
es[i].burstTime + "\t\t" + processes[i].priority + "\t\t" + processes[i].completionTime +
"\t\t" + processes[i].turnaroundTime + "\t\t" + processes[i].waitingTime);
}
```

- The output shows the following details for each process:
 - **PID:** Process ID.
 - **Arrival Time.**
 - **Burst Time.**
 - **Priority.**
 - **Completion Time.**
 - **Turnaround Time.**
 - **Waiting Time.**

6. Example Walkthrough

Input:

```
yaml
```

```
Number of processes: 3
Arrival Time: 0 1 2
Burst Time: 10 5 8
Priority: 2 1 3
```

Execution:

Step	Time	Event	Current Process	Remaining Processes
1	0	Process 1 arrives, but priority is checked.	Process 1 (Priority 2).	P2, P3
2	1	Process 2 arrives (Priority 1).	Process 2 runs.	P1, P3
3	6	Process 2 completes.	CPU idle briefly.	P1, P3
4	16	Process 1 completes.	Process 1 runs.	P3
5	24	Process 3 completes.	Process 3 runs.	-

Output:

css

PID	Arrival Time	Burst Time	Priority	Completion Time	Turnaround Time	Wait
2	1	5	1	6	5	0
1	0	10	2	16	16	6
3	2	8	3	24	22	14

Key Considerations:

- Idle CPU Handling:** When no process has arrived yet, `currentTime` increments, simulating waiting.
- Non-Preemptive Nature:** Once a process is selected, it runs to completion before another process starts.
- Priority:** Lower priority values mean higher priority execution.

Potential Improvements:

- Error Handling:** Add checks for invalid inputs (e.g., negative times or priorities).
- Preemptive Scheduling:** Modify to handle cases where higher-priority processes arrive mid-execution.
- Visualization:** Add Gantt charts to visualize scheduling progress.