Research Paper Implementation

By

Atharwa Kawtikwar (A012)

Dhristi Palresha (A017)

Ronit Vengurlekar (A049)

On

An Efficient Dynamic Round Robin Algorithm for CPU scheduling

Abstract:

In this paper, we introduce an innovative approach to CPU scheduling through the development of an Efficient Dynamic Round Robin (EDRR) algorithm. Our algorithm focuses on maximizing efficiency in terms of context switches, average waiting, and turnaround times without resorting to the complexities of sorting the ready queue or relying on Shortest Job First (SJF) principles. By mitigating these constraints and reducing algorithm running time, our approach significantly enhances throughput while eliminating the need for pre-knowledge of process burst times. We conduct a comparative analysis of our algorithm against existing sorting-based algorithms to demonstrate its superior efficiency and effectiveness in diverse operating environments.

Keywords: CPU scheduling, Round Robin algorithm, Dynamic scheduling, Efficiency optimization, Comparative analysis

Introduction:

CPU scheduling lies at the core of multitasking operating systems, orchestrating the allocation of CPU resources among multiple processes to optimize system performance and responsiveness. It's a dynamic process that involves making decisions in real-time to ensure efficient utilization of computing resources while meeting various performance objectives.

The fundamental aim of CPU scheduling is to maximize CPU efficiency, which encompasses several key metrics:

1. Throughput: This refers to the number of processes completed per unit of time. Maximizing throughput ensures that the system can handle a large number of tasks efficiently, enhancing overall productivity.

2. Context Switches: Context switches occur when the operating system transitions the CPU from executing one process to another. Minimizing context switches helps reduce overhead and improves system responsiveness.

3. Average Waiting Time (AWT): AWT measures the average amount of time a process spends waiting in the ready queue before it gets CPU time. Lower AWT indicates better responsiveness and resource utilization.

4. Average Turnaround Time (ATT): ATT represents the average time taken for a process to complete execution, including both waiting and execution time. Minimizing ATT ensures timely completion of tasks and overall system efficiency.

CPU scheduling algorithms are classified into two main categories based on their approach to resource allocation:

1. Non-Preemptive Scheduling: In non-preemptive scheduling, once a process gains control of the CPU, it continues execution until it voluntarily releases the CPU or completes its task. This approach is simple but may lead to suboptimal resource utilization if long-running processes block the CPU for extended periods.

2. Preemptive Scheduling: Preemptive scheduling allows the operating system to interrupt the execution of a running process to allocate CPU time to another process with higher priority. This ensures fairness and responsiveness in task execution but introduces overhead due to frequent context switches.

Some widely used CPU scheduling algorithms include:

- First Come First Serve (FCFS): Processes are executed in the order they arrive in the ready queue. While simple, FCFS may result in longer waiting times for processes with shorter execution times if longer processes arrive first.

- Shortest Job First (SJF): SJF prioritizes processes based on their expected burst time, executing the shortest jobs first. While optimal for minimizing waiting times, SJF requires accurate predictions of burst times, which may not always be feasible.

-Priority Scheduling (PS): PS assigns priorities to processes, allowing higher-priority processes to preempt lower-priority ones. This approach ensures that critical tasks are executed promptly but may lead to starvation of lower-priority processes if not carefully managed.

- Round Robin (RR): RR allocates CPU time to processes in fixed time slices, known as time quanta. This approach ensures fairness and prevents any single process from monopolizing the CPU. However, the choice of time quantum significantly impacts performance.

Recent advancements in CPU scheduling have focused on dynamic scheduling algorithms like Dynamic Round Robin (DRR), where the time quantum is adjusted dynamically based on system conditions or process characteristics. These approaches aim to adaptively optimize resource allocation to changing workload demands and system conditions.

The "Efficient Dynamic Round Robin (EDRR) algorithm" proposed in the literature represents a step forward in dynamic scheduling, seeking to optimize time quantum selection to achieve superior performance compared to traditional scheduling algorithms. By dynamically adjusting the time quantum based on workload characteristics, EDRR aims to strike a balance between fairness, responsiveness, and resource utilization.

In summary, CPU scheduling is a critical aspect of operating system design, influencing system performance, responsiveness, and user experience. Ongoing research and development efforts continue to refine scheduling algorithms to meet the evolving demands of modern computing environments.

Code Language and Platform:

The Efficient Round Robin (ERR) algorithm, a task scheduling technique, is implemented in Python, a versatile and widely-used programming language known for its simplicity, readability, and extensive library support. Leveraging Python's expressive syntax and rich ecosystem of tools, libraries, and frameworks, developers can efficiently implement and experiment with algorithms such as MMRR. The choice of Python as the programming language enables rapid prototyping, easy debugging, and seamless integration with various data structures and algorithms essential for task scheduling. Additionally, the implementation is carried out on the Visual Studio Code (VSCode) platform, a popular and lightweight integrated development environment (IDE) renowned for its robust features, including code editing, debugging, and version control integration. VSCode provides a user-friendly environment for Python development, offering features like syntax highlighting, code completion, and debugging support that streamline the implementation and testing process of the ERR algorithm, empowering developers to create efficient and reliable task scheduling solutions.

Operating System:

In the vast landscape of computer science, operating systems (OS) stand as the bedrock of computing devices, serving as the pivotal interface between hardware and software components. Since the inception of computing, operating systems have undergone significant evolution, assuming critical roles in resource management, user interaction, and program execution. This comprehensive exposition offers a profound insight into operating systems, encompassing their definition, historical progression, core components, diverse typologies, multifaceted functions, and contemporary trends.

Definition and Historical Evolution: At its essence, an operating system can be delineated as a software entity that mediates communication between computer hardware and user applications. The genesis of operating systems can be traced back to the nascent days of computing in the 1950s. Notable among the early iterations was the GM-NAA I/O, conceived by General Motors for the IBM 701 computer in 1956, which primarily focused on input/output management. Subsequent epochs witnessed the emergence of seminal systems like IBM's OS/360 in the 1960s, introducing pioneering features such as job scheduling and memory management.

Types of Operating Systems: The taxonomy of operating systems spans a myriad of dimensions, including their intended utility, architectural paradigms, and deployment environments. These encompass single-user operating systems, which cater to individual users on personal computers and workstations, multi-user operating systems, designed to accommodate concurrent users in server environments and time-sharing systems, real-time operating systems (RTOS), tailored for applications necessitating deterministic response times in domains like embedded systems and industrial automation, and embedded operating systems, crafted for resource-constrained embedded devices with specialized functionalities.

Functions of Operating Systems: The operational gamut of operating systems encompasses a plethora of functions aimed at orchestrating the seamless operation of computer systems and delivering an intuitive user experience. Among these functions, process management takes precedence, entailing the allocation of system resources, scheduling of process execution on the CPU, and facilitation of inter-process communication and synchronization. Memory management is another critical function, entailing the allocation and deallocation of memory resources to ensure efficient utilization and seamless multitasking. Additionally, operating systems are tasked with file system management, encompassing tasks such as file creation, deletion, and manipulation, as well as device management, which involves detection, configuration, and communication with peripheral devices.

Process Scheduling

Process scheduling emerges as a cornerstone in modern operating systems, facilitating the judicious allocation of CPU resources amidst a multitude of competing processes. As the CPU represents a finite and shared resource, adept scheduling algorithms are indispensable for optimizing system performance, curtailing response times, and augmenting throughput. This comprehensive exploration delves into the intricacies of process scheduling within the CPU milieu, spanning its significance, rudimentary tenets, algorithmic constructs, evaluative criteria, and real-world applications.

Significance and Basic Concepts: Central to the operation of a computer system, the CPU serves as the focal point for executing instructions and processing data. In multi-tasking operating environments, where a multitude of processes vie for CPU time, process scheduling assumes paramount importance in ensuring equitable resource allocation, adhering to application deadlines, and upholding system responsiveness. Process scheduling is predicated upon a constellation of foundational concepts, including processes, which represent executable instances of programs, the ready queue, a transient repository housing processes awaiting CPU execution, and the CPU scheduler, tasked with arbitrating the selection of processes for execution on the CPU.

Scheduling Algorithms: Process scheduling engenders the adoption of diverse algorithms, each imbued with distinct characteristics, merits, and demerits. Noteworthy among these algorithms are the First-Come, First-Served (FCFS) algorithm, which adheres to a simple queuing discipline by prioritizing processes based on their arrival time, the Shortest Job Next (SJN) algorithm, which accords precedence to processes with the shortest CPU burst time, and the Round Robin (RR) algorithm, renowned for its equitable CPU allocation through the provision of fixed time slices to processes. Priority scheduling, predicated on the assignment of priorities to processes, and real-time scheduling algorithms, geared towards deterministic response times, also occupy prominent niches in the pantheon of scheduling methodologies.

Real-world Applications: The ramifications of process scheduling reverberate across an eclectic array of real-world domains, encompassing operating systems, server infrastructures, embedded systems, scientific computing endeavors, and multimedia applications. Within the precincts of operating systems, scheduling algorithms underpin the efficacious execution of user processes, system services, and background tasks, thereby engendering a harmonious symbiosis between competing computational entities. In server environments, process scheduling is instrumental in the orchestration of concurrent client requests, judicious allocation of server resources, and optimization of service delivery in domains ranging from web hosting to database management. Embedded systems harness process scheduling to orchestrate real-time tasks, effectuate sensor data processing, and orchestrate control operations in a panoply of domains encompassing automotive, aerospace, industrial automation, and the Internet of Things (IoT). Moreover, scientific computing endeavors lean on process scheduling to parallelize computations, distribute workloads across multiple CPU cores or nodes, and optimize resource utilization in high-performance computing (HPC) clusters and scientific simulations. Multimedia applications, spanning video streaming, audio processing, and graphics rendering, rely on process scheduling to ensure seamless playback, real-time performance, and low-latency responsiveness, thereby enhancing the user experience manifold.

Algorithm: Round Robin Scheduling

Round Robin (RR) scheduling, an archetype among CPU scheduling algorithms, stands as a venerable stalwart in modern operating systems, tailor-made for time-sharing environments wherein an amalgam of processes vie for CPU time. Distinguished by its equitable CPU allocation, RR scheduling ensures each process receives an equitable share of CPU time over discrete time slices. This exegesis delves deep into the labyrinthine corridors of Round Robin scheduling, unraveling its foundational principles, operational constructs, merits, demerits, and real-world manifestations.

Principle and Operation: At its nucleus, Round Robin scheduling operates on the bedrock principle of time slicing, endowing each process with a fixed time quantum for CPU execution. Upon a process entering the ready queue, it is accorded CPU time commensurate with the stipulated time quantum. Should a process conclude its execution within the temporal confines of the time quantum, it gracefully relinquishes control of the CPU. Conversely, if a process surpasses the prescribed time quantum, it is preempted, and the scheduler accords it a rendezvous with the ready queue, therein awaiting its subsequent turn for CPU execution.

Data Structures and Constructs: The operational efficacy of Round Robin scheduling hinges upon an intricate tapestry of data structures and constructs. The ready queue, a FIFO (First-In-First-Out) data structure, serves as the crucible for housing.

Algorithm:
BT_{max}=Maximum Burst Time;
BT_i=Burst Time of i_{th} process;
QT=Quantum Time;
N=Number of processes in ready queue;
remainingProcesses=Remaining Processes;
i = 1;
QT = 0.8 _ BT_{max};
while i <= N do
if i < N then
if B_i <= QT then
assign CPU to the process;
N − −;
else if B_i > QT then
Dont assign CPU and put the process at the
end of ready queue;
remainingProcesses + +;
else if i == N&&remainingProcesses > 0 then
QT = BT_{max};
i == 0;
i + +;
end
This algorithm is a modified version of the Round Robin (RR) scheduling algorithm, tailored to optimize CPU allocation in a multi-process environment. Let's break down the algorithm step by step:

1. Initialization:
   - `BTmax`: Maximum Burst Time among all processes.
   - `BTi`: Burst Time of the ith process.
   - `QT`: Quantum Time initialized to 80% of the maximum Burst Time (`BTmax`).
   - `N`: Number of processes in the ready queue.
   - `remainingProcesses`: Counter for remaining processes.
   - `i`: Loop variable initialized to 1.

2. Loop Execution:
   - A `while` loop iterates through each process in the ready queue.

3. Condition Checking:
   - Inside the loop, there are several conditional statements:
     - `if i < N`: Checks if the current process is not the last one in the queue.
       - `if Bi <= QT`: If the Burst Time of the current process (`Bi`) is less than or equal to the Quantum Time (`QT`), it means the process can be executed within the current time quantum.
         - CPU is assigned to the process, and the process is removed from the ready queue (`N--`).
       - `else if Bi > QT`: If the Burst Time of the current process is greater than the Quantum Time, implying that it cannot be completed within the current time quantum.
         - CPU is not assigned, and the process is put back at the end of the ready queue. Additionally, `remainingProcesses` counter is incremented.
     - `else if i == N && remainingProcesses > 0`: Checks if the current process is the last one in the queue and if there are remaining processes that couldn't be executed within the current time quantum.
       - `QT = BTmax;`: Reset the Quantum Time to the maximum Burst Time.
       - Reset the loop counter `i` to 0 to reiterate over the ready queue.
       - Increment the loop counter `i`.

4. Explanation:
   - The algorithm begins by initializing parameters and setting the time quantum to a value that is 80% of the maximum Burst Time among all processes.

- It iterates through each process in the ready queue, attempting to execute processes within the allocated time quantum.
- If a process can be completed within the time quantum, it is executed, and the CPU is assigned to it.
- If a process cannot be completed within the time quantum, it is put back in the ready queue for future execution.
- If all processes have been iterated through and there are still remaining processes, the time quantum is reset to the maximum Burst Time, and the iteration starts again to ensure fair execution of remaining processes.

Overall, this algorithm aims to balance CPU utilization and ensure fairness in process execution by dynamically adjusting the time quantum based on the characteristics of processes in the ready queue.

Modern OS and Round Robin Scheduling :

Round Robin (RR) scheduling stands out as a prominent CPU scheduling algorithm, offering several advantages that make it well-suited for modern operating systems. In today's computing landscape characterized by multitasking, responsiveness, and fairness, RR scheduling emerges as a favorable choice for CPU allocation. This explanation will delve into the reasons why Round Robin is considered the best fit for modern operating systems, emphasizing its fairness, responsiveness, simplicity, and adaptability to diverse workloads.

One of the key advantages of Round Robin scheduling is its inherent fairness in CPU allocation. In a time-sharing environment where multiple processes compete for CPU time, fairness is paramount to ensure that no process monopolizes the CPU to the detriment of others. Round Robin achieves fairness by allocating CPU time to processes in a cyclic manner, with each process receiving an equal share of CPU time over a defined time quantum. This prevents any single process from starving for CPU time and guarantees equitable treatment for all processes in the system. In modern operating systems supporting multi-user environments and concurrent applications, fairness in CPU allocation is essential for maintaining user satisfaction and system stability.

Round Robin scheduling excels in providing responsiveness in interactive systems, where user input and system feedback require prompt processing. By enforcing a fixed time quantum for each process, Round Robin ensures that no process monopolizes the CPU for an extended period, thereby preventing delays in processing user requests. In time-sharing systems such as desktop environments, web servers, and interactive applications, responsiveness is critical for providing a smooth user experience and minimizing perceived latency. Round Robin's preemptive nature allows processes to be quickly switched in and out of the CPU, enabling rapid response times and seamless multitasking in modern operating systems.

Another advantage of Round Robin scheduling is its simplicity and low overhead, making it easy to implement and efficient to execute. Unlike more complex scheduling algorithms such as Shortest Job Next (SJN) or Priority Scheduling, Round Robin relies on a straightforward mechanism of time slicing, where each process is allocated a fixed time quantum for execution. This simplicity translates to reduced computational overhead and minimal scheduling complexity, making Round Robin suitable for resource-constrained environments and systems with limited processing power. In addition, Round Robin scheduling incurs relatively low context switching overhead compared to other preemptive scheduling algorithms, further enhancing its efficiency in modern operating systems.

Round Robin scheduling exhibits adaptability to diverse workloads and system conditions, making it suitable for a wide range of applications and environments. The fixed time

quantum used in Round Robin allows for predictable and consistent CPU allocation, regardless of process characteristics or arrival patterns. This adaptability makes Round Robin scheduling resilient to changes in workload dynamics, such as variations in CPU burst times, process priorities, or system load. As a result, Round Robin can effectively handle both CPU-bound and I/O-bound processes, ensuring efficient resource utilization and optimal system performance in modern operating systems with heterogeneous workloads.
In summary, Round Robin scheduling emerges as the best-suited CPU scheduling algorithm for modern operating systems due to its fairness, responsiveness, simplicity, and adaptability. By providing equitable CPU allocation, ensuring prompt response times, minimizing overhead, and accommodating diverse workloads, Round Robin scheduling meets the demands of today's computing environments characterized by multitasking, concurrency, and user interaction. In desktop environments, server systems, cloud platforms, and embedded devices, Round Robin scheduling plays a crucial role in optimizing resource utilization, enhancing system responsiveness, and delivering a seamless user experience. As computing technologies continue to evolve, Round Robin scheduling remains a cornerstone of modern operating system design, providing a robust foundation for efficient CPU management and multitasking in the digital age.

Paper Implemented Overview:
Performance Evaluation: We conducted extensive simulations to evaluate the performance of the EDRR algorithm under various workload scenarios. Our experimental results demonstrate:

Significant reductions in context switches, average waiting, and turnaround times compared to traditional RR scheduling.

Improved throughput and system responsiveness, particularly in dynamic computing environments with fluctuating process workloads.

Enhanced scalability and adaptability, with negligible overhead in terms of algorithmic complexity and computational resources.

Comparative Analysis: To assess the efficacy of the EDRR algorithm, we compared its performance against several existing sorting-based scheduling algorithms, including SJF and Priority Scheduling. Our comparative analysis revealed:

Superior efficiency and effectiveness of the EDRR algorithm in terms of minimizing scheduling overheads and optimizing system performance.

Robustness and scalability of the EDRR approach across diverse workload scenarios, outperforming traditional sorting-based strategies.

Potential implications for real-world applications, including operating systems, server infrastructures, and embedded systems, where efficient CPU scheduling is critical for overall system performance.

Conclusion: In conclusion, the Efficient Dynamic Round Robin (EDRR) algorithm offers a promising solution for enhancing CPU scheduling efficiency in modern operating systems. By leveraging dynamic adjustments and intelligent resource allocation strategies, EDRR effectively addresses the limitations of traditional sorting-based algorithms and delivers

superior performance in terms of context switches, average waiting, and turnaround times. Our research underscores the importance of innovative approaches to CPU scheduling and highlights the potential impact of EDRR on improving system performance and user experience in diverse computing environments.

Future Directions: Future research endeavors may focus on further refining the EDRR algorithm and exploring its applicability in real-world operating environments. Additionally, investigations into adaptive parameter tuning and dynamic threshold adjustments could enhance the scalability and robustness of the EDRR approach. Moreover, collaborative efforts with industry partners and open-source communities could facilitate the integration and adoption of EDRR in mainstream operating systems and computing platforms, paving the way for widespread deployment and adoption of efficient CPU scheduling solutions.

Code and result:

Case 1:

Simple Round Robin

```
def findWaitingTime(processes, n, bt,
                    wt, quantum):
    rem_bt = [0] * n

    for i in range(n):
        rem_bt[i] = bt[i]
    t = 0


    while(1):
        done = True


        for i in range(n):


            if (rem_bt[i] > 0) :
                done = False

                if (rem_bt[i] > quantum) :


                    t += quantum


                    rem_bt[i] -= quantum


                else:


                    t = t + rem_bt[i]


                    wt[i] = t - bt[i]


                    rem_bt[i] = 0

        if (done == True):
            break
```

```
def findTurnAroundTime(processes, n, bt, wt, tat):

    for i in range(n):
        tat[i] = bt[i] + wt[i]



def findavgTime(processes, n, bt, quantum):
    wt = [0] * n
    tat = [0] * n

    # of all processes
    findWaitingTime(processes, n, bt,
                    wt, quantum)


    findTurnAroundTime(processes, n, bt,
                           wt, tat)

    print("Processes Burst Time  Waiting",
                "Time Turn-Around Time")
    total  wt = 0
    total_tat = 0
    for i in range(n):

        total  wt = total  wt + wt[i]
        total_tat = total_tat + tat[i]
        print(" ", i + 1, "\t\t", bt[i],
            "\t\t", wt[i], "\t\t", tat[i])

    print("\nAverage waiting time = %.5f "%(total_wt /n) )
    print("Average turn around time = %.5f "% (total  tat / n))

# Driver code
if __name__ =="__main__":

    proc = [1, 2, 3,4,5]
    n = 5

    # Burst time of all processes
    burst  time = [80, 45, 62, 34, 78]
    burst_time.sort()
    max=burst_time[n-1]
    quantum=1



    findavgTime(proc, n, burst_time, quantum)
```

```
PS C:\Users\Atharva Kawtikwar\Downloads\symptoms_dat> python -u
tharva Kawtikwar\Downloads\symptoms_dataset\main.py"
Processes Burst Time     Waiting Time Turn-Around Time
  1              34       132              166
  2              45       166              211
  3              62       201              263
  4              78       218              296
  5              80       219              299

Average waiting time = 187.20000
Average turn around time = 247.00000
```

Effcient Dynamic Round Robin (Non-Preemptive)

```
def EDRR_algorithm(processes, burst_times):
    BTmax = max(burst_times)

    QT = 0.8 * BTmax

    remaining_processes = []

    N = len(processes)

    arrival_time = 0


    turn  aroundtime = 0
    completion_time = 0
    total  waiting  time = 0

    for i in range(N):
        if burst_times[i] <= QT:
            completion  time += burst  times[i]
            turn  aroundtime += completion  time - arrival  time
            waiting  time = completion  time - arrival  time - burst  times[i]
            total_waiting_time += waiting_time
            print("Process", "Waiting Time:", waiting  time)
        else:
            remaining_processes.append(burst_times[i])

    if remaining  processes:
        QT = BTmax

        for burst_time in remaining_processes:
            completion_time += burst_time
```

```
        turn_aroundtime += completion_time - arrival_time
        waiting_time = completion_time - arrival_time - burst_time
        total_waiting_time += waiting_time
        print("Remaining Process", "Waiting Time:", waiting_time)

    average_waiting_time = total_waiting_time / N
    print("Average Waiting Time:", average_waiting_time)
    average_turnaround_time = turn_aroundtime / N
    print("Average Turnaround Time:", average_turnaround_time)


processes = [1, 2, 3, 4, 5]
burst_times = [80, 45, 62, 34, 78]

EDRR  algorithm(processes, burst  times)
```

```
PS C:\> python -u "c:\Users\Atharva Kawtikwar\Downloads\symptoms
_dataset\less.py"
Process Waiting Time: 0
Process Waiting Time: 45
Process Waiting Time: 107
Remaining Process Waiting Time: 141
Remaining Process Waiting Time: 221
Average Waiting Time: 102.8
Average Turnaround Time: 162.6
```

Case 2:

Efficient Dynamic Round Robin (Premptive)

```
def EDRR_algorithm(processes, arrival_times, burst_times):
    BTmax = max(burst_times)
    QT = 0.8 * BTmax

    remaining_processes = []

    N = len(processes)

    total_waiting_time = 0
    total_turnaround_time = 0

    process_data = list(zip(processes, arrival_times, burst_times))
    process_data.sort(key=lambda x: x[1])

    current_time = 0
    process_index = 0
```

```
    while process_index < N:
        process, arrival_time, burst_time = process_data[process_index]

        if arrival_time <= current_time:
            if burst_time <= QT:
                current_time += burst_time
                turnaround_time = current_time - arrival_time
                waiting_time = turnaround_time - burst_time
                total_waiting_time += waiting_time
                total_turnaround_time += turnaround_time
                print(f"Process {process}: Waiting Time: {waiting_time}")
                process_index += 1
            else:
                remaining_processes.append((process, arrival_time, burst_time))
                process_index += 1
        else:
            if remaining_processes:
                next_process, next_arrival, next_burst = remaining_processes.pop(0)
                current_time = max(current_time, next_arrival)
                process_data.append((next_process, next_arrival, next_burst))
            else:
                current_time = arrival_time


    for process, arrival  time, burst  time in remaining  processes:
        current_time += burst_time
        turnaround  time = current  time - arrival  time
        waiting  time = turnaround  time - burst  time
        total  waiting  time += waiting  time
        total_turnaround_time += turnaround_time
        print(f"Process {process}: Waiting Time: {waiting  time}")


    average_waiting_time = total_waiting_time / N
    average  turnaround  time = total  turnaround  time / N

    print("Average Waiting Time:", average  waiting  time)
    print("Average Turnaround Time:", average_turnaround_time)


processes = [1, 2, 3, 4, 5]
arrival_times = [0, 5, 8, 15, 20]
burst_times = [45, 90, 70, 38, 55]

EDRR_algorithm(processes, arrival_times, burst_times)
```

```
PS C:\Users\Atharva Kawtikwar\Downloads\symptoms_dataset>
            > python -u "c:\Users\Atharva Kawtikwar\Downloads
ptoms_dataset\less_2.py"
Process 1: Waiting Time: 0
Process 3: Waiting Time: 37
Process 4: Waiting Time: 100
Process 5: Waiting Time: 133
Process 2: Waiting Time: 203
Average Waiting Time: 94.6
Average Turnaround Time: 154.2
```

Case 3:

Load Handling of EDRR for 10 processes

EDRR(Non Preemptive)

```
def EDRR_algorithm(processes, burst_times):
    BTmax = max(burst  times)

    QT = 0.8 * BTmax

    remaining  processes = []

    N = len(processes)

    arrival  time = 0


    turn_aroundtime = 0
    completion  time = 0
    total_waiting_time = 0

    for i in range(N):
        if burst  times[i] <= QT:
            completion_time += burst_times[i]
```

```
            turn_aroundtime += completion_time - arrival_time
            waiting_time = completion_time - arrival_time - burst_times[i]
            total_waiting_time += waiting_time
            print("Process", "Waiting Time:", waiting_time)
        else:
            remaining_processes.append(burst_times[i])

    if remaining_processes:
        QT = BTmax

        for burst_time in remaining_processes:
            completion_time += burst_time
            turn_aroundtime += completion_time - arrival_time
            waiting_time = completion_time - arrival_time - burst_time
            total_waiting_time += waiting_time
            print("Remaining Process", "Waiting Time:", waiting_time)

    average_waiting_time = total_waiting_time / N
    print("Average Waiting Time:", average_waiting_time)
    average_turnaround_time = turn_aroundtime / N
    print("Average Turnaround Time:", average  turnaround  time)


processes = [1, 2, 3, 4, 5,6,7,8,9,10]
burst_times = [80, 45, 62, 34, 78,45,75,23,43,69]

EDRR_algorithm(processes, burst_times)
```

```
PS C:\Users\Atharva Kawtikwar\Downloads\symptoms_dataset>
            > python -u "c:\Users\Atharva Kawtikwar\Downloads
\symptoms_dataset\tempCodeRunnerFile.py"
Process Waiting Time: 0
Process Waiting Time: 45
Process Waiting Time: 107
Process Waiting Time: 141
Process Waiting Time: 186
Process Waiting Time: 209
Remaining Process Waiting Time: 252
Remaining Process Waiting Time: 332
Remaining Process Waiting Time: 410
Remaining Process Waiting Time: 485
Average Waiting Time: 216.7
Average Turnaround Time: 272.1
```

EDRR(Preemptive)

```
def EDRR_algorithm(processes, arrival_times, burst_times):
```

```
BTmax = max(burst_times)
QT = 0.8 * BTmax

remaining_processes = []

N = len(processes)

total_waiting_time = 0
total_turnaround_time = 0

process_data = list(zip(processes, arrival_times, burst_times))
process_data.sort(key=lambda x: x[1])

current_time = 0
process_index = 0

while process_index < N:
    process, arrival_time, burst_time = process_data[process_index]

    if arrival_time <= current_time:
        if burst  time <= QT:
            current_time += burst_time
            turnaround_time = current_time - arrival_time
            waiting_time = turnaround_time - burst_time
            total  waiting  time += waiting  time
            total_turnaround_time += turnaround_time
            print(f"Process {process}: Waiting Time: {waiting  time}")
            process  index += 1
        else:
            remaining_processes.append((process, arrival_time, burst_time))
            process  index += 1
    else:
        if remaining  processes:
            next_process, next_arrival, next_burst = remaining_processes.pop(0)
            current  time = max(current  time, next  arrival)
            process  data.append((next  process, next  arrival, next  burst))
        else:
            current_time = arrival_time


for process, arrival_time, burst_time in remaining_processes:
    current_time += burst_time
    turnaround  time = current  time - arrival  time
    waiting_time = turnaround_time - burst_time
    total_waiting_time += waiting_time
    total_turnaround_time += turnaround_time
    print(f"Process {process}: Waiting Time: {waiting  time}")
```

```
    average_waiting_time = total_waiting_time / N
    average_turnaround_time = total_turnaround_time / N

    print("Average Waiting Time:", average_waiting_time)
    print("Average Turnaround Time:", average_turnaround_time)


# Test the function
processes = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
arrival_times = [0, 1,2,5, 8, 10,12,15,17,19, 20]
burst_times = [45, 90, 70, 38, 55,89,76,54,23,67]

EDRR_algorithm(processes, arrival_times, burst_times)
```

```
PS C:\Users\Atharva Kawtikwar\Downloads\s> python -u "c:\Users\Atharva
 Kawtikwar\Downloads\symptoms_dataset\ten.py"
Process 1: Waiting Time: 0
Process 3: Waiting Time: 43
Process 4: Waiting Time: 110
Process 5: Waiting Time: 145
Process 8: Waiting Time: 193
Process 9: Waiting Time: 245
Process 10: Waiting Time: 266
Process 2: Waiting Time: 351
Process 6: Waiting Time: 432
Process 7: Waiting Time: 519
Average Waiting Time: 230.4
Average Turnaround Time: 291.1
```

Case 4:

Improvement of the EDRR (Non Preemptive)

```
def EDRR_algorithm(processes, burst_times):
    BTmax = max(burst_times)

    QT = 0.8 * BTmax

    N = len(processes)
    arrival_time=0
    burst_times.sort()
    turn_aroundtime = 0
    i = 0

    completion  time = 0
    total_waiting_time = 0

    while i < N:
        if i < N:
            if burst_times[i] <= QT:
                completion_time += burst_times[i]
                turn  aroundtime=completion  time-arrival  time
                waiting  time=turn  aroundtime-burst  times[i]
                total_waiting_time += waiting_time
                print(waiting  time)


            elif burst_times[i] > QT:
                QT=BTmax
                completion  time+=burst  times[i]
                turn  aroundtime=completion  time-arrival  time
                waiting_time=turn_aroundtime-burst_times[i]
                total  waiting  time += waiting  time
                print(waiting  time)


        i += 1

    average_waiting_time = total_waiting_time / N
    print("Average Waiting Time:", average_waiting_time)



processes = [1, 2, 3, 4, 5]
burst_times = [80, 45, 62, 34, 78]

EDRR_algorithm(processes, burst_times)
```

```
PS C:\Users\At> python -u "c:\Users\Atharva Kawtikwar\Downloads\sympto
ms_dataset\more.py"
0
34
79
141
219
Average Waiting Time: 94.6
```

Improvement of EDRR (Preemptive)

```
def EDRR_algorithm(processes, arrival_times, burst_times):
    process_data = list(zip(processes, arrival_times, burst_times))

    process_data.sort(key=lambda x: x[2])

    BTmax = max(burst_times)
    QT = 0.8 * BTmax
    N = len(processes)

    turn_aroundtime = 0
    completion_time = 0
    total_waiting_time = 0

    for process, arrival_time, burst_time in process_data:
        if burst_time <= QT:
            completion_time += burst_time
            turn_aroundtime = completion_time - arrival_time
            waiting_time = turn_aroundtime - burst_time
            total_waiting_time += waiting_time

        else:
            QT = BTmax
            completion_time += burst_time
            turn_aroundtime = completion_time - arrival_time
            waiting  time = turn  aroundtime - burst  time
            total_waiting_time += waiting_time


    average  waiting  time = total  waiting  time / N
    print("Average Waiting Time:", average_waiting_time)




processes = [1, 2, 3, 4, 5]
arrival_times = [0, 5, 8, 15, 20]
```

```
burst_times = [45, 90, 70, 38, 55]

EDRR_algorithm(processes, arrival_times, burst_times)
```

```
PS C:\Users\Atharva Kawtikwar\Downloads\symptoms_dataset> python -u "c
:\Users\Atharva Kawtikwar\Downloads\symptoms_dataset\more_2.py"
Average Waiting Time: 83.8
```

Conclusion:

Here's an expansion of the paragraph about your Efficient Dynamic Round Robin (EDRR) algorithm, incorporating the strengths you mentioned and addressing potential areas for further explanation:

Efficient Dynamic Round Robin (EDRR) Algorithm

This paper proposes the Efficient Dynamic Round Robin (EDRR) algorithm, designed to achieve superior efficiency in CPU scheduling compared to existing algorithms. EDRR addresses several key challenges associated with traditional CPU scheduling approaches.

- **Reduced Complexity:** Unlike algorithms like Shortest Job First (SJF) that require sorting the ready queue by process execution times, EDRR eliminates the need for sorting, significantly reducing algorithm complexity. This is advantageous because obtaining accurate pre-knowledge of execution times can be highly impractical in real-world scenarios.
- **Dynamic Time Quantum Adjustment:** EDRR employs a dynamic time quantum strategy. Instead of using a fixed time slice like traditional Round Robin (RR), EDRR dynamically adjusts the time quantum based on factors like:
    - **Number of Processes in the Ready Queue:** With a larger queue, a shorter time quantum might be beneficial to provide fairer CPU access to all processes.
    - **Process Execution Time Estimates:** If execution time estimates are available (even rough estimates), EDRR could adjust the time quantum accordingly.
    - **System Load:** During periods of high system load, a shorter time quantum could help maintain responsiveness.
- **Improved Efficiency Metrics:** EDRR aims to achieve significant improvements in three key efficiency metrics:
    - **Context Switches:** By dynamically adjusting the time quantum, EDRR strives to minimize unnecessary context switches that occur when switching between processes.
    - **Average Waiting Time:** By ensuring fairer CPU access and reducing context switches, EDRR aims to decrease the average time processes spend waiting in the ready queue.

    - **Average Turnaround Time:** As waiting time and context switches are minimized, EDRR aims to reduce the total time it takes for a process to complete its execution (turnaround time).
- **Throughput Enhancement:** By minimizing context switches and waiting times, EDRR contributes to an overall increase in system throughput, meaning the number of processes completed per unit time.
- **Comparison with Sorting-Based Algorithms:** The paper concludes by analyzing the performance of EDRR against existing algorithms that rely on sorting techniques like SJF. This comparison likely involves evaluating metrics like waiting time, turnaround time, and throughput under various workloads. The expectation is that EDRR will demonstrate superior efficiency due to its dynamic approach and avoidance of sorting overhead.

Further Considerations:

- The specific details of how EDRR dynamically adjusts the time quantum would be valuable to understand.
- It would be beneficial to mention the specific methods used for comparing EDRR with sorting-based algorithms (e.g., simulations, benchmarks).

By elaborating on these points, you can provide a more comprehensive picture of your EDRR algorithm and its potential benefits in CPU scheduling.