WAS Scheduling Algorithm

Project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology

In

Computer Science and Engineering

School of Engineering and Sciences

Submitted by

P Sushanth AP23110010989
P Chashmitha AP23110011087
Rishi Kumar AP23110011088
M Reethu Priya AP23110011128
M Olive Chaitanya AP23110011192

Under the Guidance of

Dr. Subhankar Ghatak

SRM University–AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh – 522 240

[November, 2025]

Certificate

Date: 16-Nov-25

This is to certify that the project titled **"WAS Scheduling Algorithm"** has been undertaken and completed by **P. Sushanth, P. Chashmitha, Rishi Kumar, M. Reethu Priya, and M. Olive Chaitanya** under my guidance. The work embodied in this project is **original, authentic**, and reflects the sincere efforts of the students. I affirm that the project meets the academic standards and is **appropriate for submission** to **SRM University – AP**, School of Engineering and Sciences, in partial fulfillment of the requirements for the **Bachelor of Technology** degree.

Supervisor

Dr. Subhankar Ghatak

Assistant Professor.

Department of Computer Science.

TABLE OF CONTENTS

S.No.	Title	Pg.No.
1	**Abstract**	**1**
2	**Problem Statement**	**1**
3	**Introduction**	**2**
4	**Objectives of this Project**	**3**
5	**Overview of Traditional CPU Scheduling Methods**	**4**
6	**Proposed Method (WAS)**	**5**
7	**System Design**	**6-7**
8	**Pseudocode**	**8**
9	**Implementation**	**9-19**
10	**Experimental Results**	**20-22**
11	**Conclusion**	**23**

ABSTRACT

Efficient CPU scheduling plays a crucial role in achieving high system performance in modern operating systems. Traditional CPU scheduling algorithms such as Shortest Job First (SJF), Shortest Remaining Time First (SRTF), and Highest Response Ratio Next (HRRN) have been widely used due to their simplicity and effectiveness. However, each algorithm suffers from significant drawbacks: SJF and SRTF face severe starvation issues for long-burst processes, while HRRN is non-preemptive and somewhat computationally heavier, making it less suitable for real-time responsiveness.

To address these limitations, this project introduces a new scheduling algorithm named **Weighted Aging Shortest (WAS)**. WAS is a preemptive algorithm that integrates the concept of **aging** into burst-based scheduling. It continuously computes a dynamic *score* for each ready process based on remaining CPU burst and waiting time. The formula ensures that long-waiting processes gradually gain higher priority, effectively eliminating starvation while still maintaining the efficiency advantages of short-job scheduling.

This report presents the detailed working, design, pseudocode, implementation, examples, performance analysis, and a comparison of WAS with SJF, SRTF, and HRRN. Experimental results demonstrate that WAS delivers near-optimal average turnaround time like SRTF and achieves fairness comparable to HRRN, making it a balanced and robust scheduling technique.

PROBLEM STATEMENT

To develop a novel CPU scheduling algorithm for a uniprocessor system that overcomes starvation and unfairness of existing burst-based scheduling algorithms while maintaining low average waiting and turnaround time. The algorithm must be simple to implement, preemptive, and tunable to fit different workloads.

INTRODUCTION

CPU scheduling is one of the core functionalities of any operating system, responsible for determining which process should be executed at any given time. Since the CPU is the most valuable and limited resource, the efficiency of process scheduling directly affects overall system performance.

Different scheduling algorithms are designed with different goals in mind, such as minimizing waiting time, reducing turnaround time, maximizing CPU utilization, avoiding starvation, or improving fairness among processes. However, no single algorithm satisfies all objectives simultaneously.

In traditional operating systems:

- FCFS suffers from the convoy effect.
- SJF performs extremely well in average waiting time but causes starvation.
- SRTF improves responsiveness but increases context switching.
- HRRN avoids starvation but is not preemptive.

This motivates the need for hybrid or improved algorithms.

Why improve CPU scheduling?

- Increasing number of processes
- Real-time responsiveness requirements
- Performance bottlenecks
- Demand for fairness
- Complexity of modern workloads

Thus, there is a need for an algorithm that:

1. Performs well in average waiting/turnaround time
2. Reduces starvation
3. Remains easy to implement
4. Maintains responsiveness through preemption

Weighted Aging Shortest (WAS) is proposed to fulfil these requirements.

OBJECTIVES OF THE PROJECT

The project sets forth the following detailed objectives:

Primary Objectives

1. To study and analyze existing CPU scheduling algorithms in detail.
2. To identify the specific shortcomings of SJF, SRTF, and HRRN.
3. To develop a new CPU scheduling algorithm that reduces starvation while maintaining performance.
4. To introduce dynamic aging into burst-based scheduling.

Secondary Objectives

5. To design WAS in a modular and easy-to-understand way for beginners.
6. To simulate WAS along with standard algorithms using the same dataset.
7. To analyze the statistical performance using turnaround time, waiting time, and Gantt charts.
8. To compare the proposed algorithm scientifically with existing algorithms.
9. To generate a clear academic report and presentation of findings.
10. To provide a clean and readable C implementation.

Overview of Traditional CPU Scheduling Methods

This explains the foundation of classical CPU scheduling systems, providing context for why **WAS** is necessary.

1. First Come First Serve (FCFS)

- Non-preemptive
- Easy to implement
- Poor with variance in job sizes
- Convoy effect occurs when a long job blocks shorter jobs
- Not suitable for interactive environments

2. Shortest Job First (SJF)

- Non-preemptive
- Processes with shortest burst are executed first
- Mathematically proven to minimize average waiting time
  Disadvantage:
- Severe starvation of long processes
- Requires exact knowledge of CPU burst times

3. Shortest Remaining Time First (SRTF)

- Preemptive version of SJF
- Continuously compares remaining burst times
- Best average turnaround time
  Disadvantages:
- Highest starvation rate
- Highest context switching overhead
- Requires prediction of CPU bursts

4. Highest Response Ratio Next (HRRN)

- Non-preemptive
- Uses formula: (Waiting + Burst) / Burst
- Favours long-waiting processes
- Eliminates starvation
  Disadvantages:
- Non-preemptive → delays responsiveness
- Response ratio recalculation is computationally heavier

PROPOSED SYSTEM – WEIGHTED AGING SHORTEST (WAS)

WAS is designed as a **preemptive**, **aging-based**, **burst-aware** scheduler.

Fundamental Idea

Use a dynamic score formula:

Score = RemainingBurst / (1 + k × WaitingTime)

Where,

- RemainingBurst decreases as process executes
- WaitingTime increases the longer the process waits
- k is an adjustable constant (0.2–1.5)

Interpretation of Score

- A short job → small numerator → low score → high priority
- Long-waiting job → high denominator → low score → boosted priority

Thus, **WAS** balances short-job preference and starvation prevention.

Key Features

1. **Preemptive** – reacts quickly to new arrivals
2. **Starvation-free** – aging eventually dominates
3. **Tunable** – change k value for fairness/performance tradeoff
4. **Simple** – uses only division and addition
5. **Efficient** – O(n) per scheduling event

Why WAS is Novel

- Combines aging with burst-based scheduling in a mathematically clean way
- Unlike HRRN, it is fully preemptive
- Unlike SRTF, it avoids starvation
- Unlike SJF, it considers waiting time dynamically

SYSTEM DESIGN

Components

- Ready Queue
- CPU Execution Unit
- Score Calculator
- Preemption Handler
- Completion Tracker

Working Cycle

At every decision point:

1. Identify all ready processes
2. Compute waiting time for each
3. Apply the score formula
4. Choose process with minimum score
5. Preempt if necessary
6. Execute 1 time unit
7. Update time, remaining burst, waiting times

Flowchart

Start

↓

Load Processes

↓

time = 0

↓

While (processes left):

↓

Find all processes with arrival <= time

↓

If none ready: Idle CPU, time++

↓

Calculate score for each ready process

↓

Choose min score process

↓

Execute for 1 unit

↓

Update rem_burst, waiting time

↓

If process finished → record completion

↓

time++

↓

End

PSEUDOCODE

```
Initialize all processes with arrival time, burst time, remaining time
Set time = 0

Repeat until all processes are complete:
    ready_set = all processes with arrival <= time and remaining > 0

    If ready_set is empty:
        time = time + 1
        continue

    For each process P in ready_set:
        waiting_time[P] = time - arrival[P]
        score[P] = remaining_time[P] / (1 + k * waiting_time[P])

    Choose process X with minimum score
    Execute X for 1 time unit
    remaining_time[X]--

    If remaining_time[X] == 0:
        completion_time[X] = time + 1

    time = time + 1
End Repeat
```

IMPLEMENTATION

Programming Language

- C language
- Simple loops, arrays, and if-conditions
- No advanced data structures

Modules

- Input module
- Core scheduling loop
- Score calculation
- Gantt chart generator
- Output and metrics module

Codes

SJF (Non-preemptive)

```
#include <stdio.h>
#include <stdlib.h>

#define N 5
#define INF 1000000000

int pid[N]    = {1,2,3,4,5};
int arrival[N]= {0,1,2,3,4};
int burst[N]  = {8,4,9,5,2};

int main() {
    int remaining[N];
    int completed = 0, time = 0;
    int ct[N]; for (int i=0;i<N;i++) ct[i]=0;
    int gantt[1000]; for(int i=0;i<1000;i++) gantt[i]=0;
```

```
for (int i=0;i<N;i++) remaining[i] = burst[i];


while (completed < N) {
    int idx = -1;
    int minburst = INF;
    for (int i=0;i<N;i++) {
        if (arrival[i] <= time && remaining[i] > 0) {
            if (burst[i] < minburst) {
                minburst = burst[i];
                idx = i;
            } else if (burst[i] == minburst && idx != -1) {
                // tie-breaker: longer waiting
                int waitA = time - arrival[i];
                int waitB = time - arrival[idx];
                if (waitA > waitB) idx = i;
            }
        }
    }
    if (idx == -1) {
        gantt[time] = 0; // idle
        time++;
        continue;
    }
    for (int t=0; t<remaining[idx]; t++) {
        gantt[time] = pid[idx];
        time++;
    }
    remaining[idx] = 0;
    ct[idx] = time;
```

```
        completed++;
    }

    printf("SJF (Non-preemptive) Gantt:\n");
    for (int t=0; t<time; t++) {
        if (gantt[t]==0) printf("| idle ");
        else printf("| P%d ", gantt[t]);
    }
    printf("|\n\n");

    int tat_sum=0, wt_sum=0;
    printf("PID\tArrival\tBurst\tCT\tTAT\tWT\n");
    for (int i=0;i<N;i++) {
        int tat = ct[i] - arrival[i];
        int wt  = tat - burst[i];
        tat_sum += tat; wt_sum += wt;
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid[i], arrival[i], burst[i], ct[i], tat, wt);
    }
    printf("Avg TAT = %.2f\n", (double)tat_sum / N);
    printf("Avg WT  = %.2f\n", (double)wt_sum / N);
    return 0;
}
```

SRTF (Preemptive)

```
#include <stdio.h>
#include <stdlib.h>

#define N 5
#define INF 1000000000
```

```
int pid[N]    = {1,2,3,4,5};
int arrival[N]= {0,1,2,3,4};
int burst[N]  = {8,4,9,5,2};

int main() {
    int rem[N];
    int ct[N]; for (int i=0;i<N;i++) ct[i]=0;
    int gantt[1000]; for(int i=0;i<1000;i++) gantt[i]=0;
    for (int i=0;i<N;i++) rem[i]=burst[i];

    int completed = 0, time = 0;
    while (completed < N) {
        int idx = -1;
        int minrem = INF;
        for (int i=0;i<N;i++) {
            if (arrival[i] <= time && rem[i] > 0) {
                if (rem[i] < minrem) {
                    minrem = rem[i];
                    idx = i;
                } else if (rem[i] == minrem && idx != -1) {
                    int waitA = time - arrival[i];
                    int waitB = time - arrival[idx];
                    if (waitA > waitB) idx = i;
                }
            }
        }
        if (idx == -1) {
            gantt[time] = 0;
            time++;
```

```
        continue;
    }
    gantt[time] = pid[idx];
    rem[idx]--;
    time++;
    if (rem[idx] == 0) {
        ct[idx] = time;
        completed++;
    }
}

printf("SRTF (Preemptive) Gantt:\n");
for (int t=0; t<time; t++) {
    if (gantt[t]==0) printf("| idle ");
    else printf("| P%d ", gantt[t]);
}
printf("|\n\n");

int tat_sum=0, wt_sum=0;
printf("PID\tArrival\tBurst\tCT\tTAT\tWT\n");
for (int i=0;i<N;i++) {
    int tat = ct[i] - arrival[i];
    int wt  = tat - burst[i];
    tat_sum += tat; wt_sum += wt;
    printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid[i], arrival[i], burst[i], ct[i], tat, wt);
}
printf("Avg TAT = %.2f\n", (double)tat_sum / N);
printf("Avg WT  = %.2f\n", (double)wt_sum / N);
return 0;}
```

HRRN (Non-preemptive)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 5

int pid[N]    = {1,2,3,4,5};
int arrival[N]= {0,1,2,3,4};
int burst[N]  = {8,4,9,5,2};

int main() {
    int rem[N], done[N], ct[N];
    int gantt[1000]; for(int i=0;i<1000;i++) gantt[i]=0;
    for (int i=0;i<N;i++) { rem[i]=burst[i]; done[i]=0; ct[i]=0; }

    int completed = 0, time = 0;
    while (completed < N) {
        int idx = -1;
        double bestRR = -1.0;
        for (int i=0;i<N;i++) {
            if (!done[i] && arrival[i] <= time) {
                int waiting = time - arrival[i];
                double rr = ((double)waiting + (double)burst[i]) / (double)burst[i];
                if (rr > bestRR) {
                    bestRR = rr; idx = i;
                } else if (fabs(rr - bestRR) < 1e-9 && idx != -1) {
                    if (waiting > time - arrival[idx]) idx = i;
                }
```

```
        }
    }
    if (idx == -1) {
        gantt[time] = 0;
        time++;
        continue;
    }
    for (int t=0; t<burst[idx]; t++) {
        gantt[time] = pid[idx];
        time++;
    }
    done[idx] = 1;
    ct[idx] = time;
    completed++;
}

printf("HRRN (Non-preemptive) Gantt:\n");
for (int t=0; t<time; t++) {
    if (gantt[t]==0) printf("| idle ");
    else printf("| P%d ", gantt[t]);
}
printf("|\n\n");

int tat_sum=0, wt_sum=0;
printf("PID\tArrival\tBurst\tCT\tTAT\tWT\n");
for (int i=0;i<N;i++) {
    int tat = ct[i] - arrival[i];
    int wt  = tat - burst[i];
    tat_sum += tat; wt_sum += wt;
```

```
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid[i], arrival[i], burst[i], ct[i], tat, wt);
    }
    printf("Avg TAT = %.2f\n", (double)tat_sum / N);
    printf("Avg WT  = %.2f\n", (double)wt_sum / N);
    return 0;
}
```

WAS (Preemptive)

```
#include <stdio.h>
#include <float.h> // For DBL_MAX
#include <math.h>  // For fabs()

#define N 5
#define K 0.5

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int completionTime;
    int isCompleted; // Flag
};

int main() {
    struct Process proc[N] = {
        {1, 0, 8, 8, 0, 0},
        {2, 1, 4, 4, 0, 0},
        {3, 2, 9, 9, 0, 0},
        {4, 3, 5, 5, 0, 0},
        {5, 4, 2, 2, 0, 0}
```

```
};

int time = 0;
int completedCount = 0;

while (completedCount < N) {
    int bestIdx = -1;
    double minScore = DBL_MAX;

    for (int i = 0; i < N; i++) {
        if (proc[i].arrivalTime <= time && proc[i].isCompleted == 0) {

            int timeSpentRunning = proc[i].burstTime - proc[i].remainingTime;
            int waitingTime = time - proc[i].arrivalTime - timeSpentRunning;
            if (waitingTime < 0) waitingTime = 0;

            double score = (double)proc[i].remainingTime / (1.0 + K * waitingTime);

            if (score < minScore) {
                minScore = score;
                bestIdx = i;
            }
            else if (fabs(score - minScore) < 1e-9) {
                if (proc[i].remainingTime < proc[bestIdx].remainingTime) {
                    bestIdx = i;
                } else if (proc[i].remainingTime == proc[bestIdx].remainingTime &&
                        proc[i].pid < proc[bestIdx].pid) {
                    bestIdx = i;
                }
            }
```

```
        }
    }

    if (bestIdx == -1) {
        time++;
        continue;
    }
    proc[bestIdx].remainingTime--;
    time++; // Advance time

    if (proc[bestIdx].remainingTime == 0) {
        proc[bestIdx].completionTime = time;
        proc[bestIdx].isCompleted = 1;
        completedCount++;
    }
}

printf("PID\tArrival\tBurst\tCT\tTAT\tWT\n");
printf("-------------------------------------------------\n");

double totalTAT = 0, totalWT = 0;

for (int i = 0; i < N; i++) {
    int tat = proc[i].completionTime - proc[i].arrivalTime;
    int wt = tat - proc[i].burstTime;
    totalTAT += tat;
    totalWT += wt;

    printf("P%d\t%d\t%d\t%d\t%d\t%d\n",
           proc[i].pid, proc[i].arrivalTime, proc[i].burstTime,
```

```
            proc[i].completionTime, tat, wt);
    }

    printf("-------------------------------------------------\n");
    printf("Avg TAT = %.2f\n", totalTAT / N);
    printf("Avg WT  = %.2f\n", totalWT / N);

    return 0;
}
```

EXPERIMENTAL RESULTS

Input Dataset

PID	Arrival	Burst
1	0	8
2	1	4
3	2	9
4	3	5
5	4	2

Gantt Charts

SJF (Non-Preemtive)

```
SJF (Non-preemptive) Gantt:
| P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P5 | P5 | P2 | P2 | P2 | P2 | P4 | P4 | P4 |
P4 | P4 | P3 | P3 | P3 | P3 | P3 | P3 | P3 | P3 | P3 |

PID  Arrival   Burst     CT   TAT  WT
1    0    8    8    8    0
2    1    4    14   13   9
3    2    9    28   26   17
4    3    5    19   16   11
5    4    2    10   6    4
Avg TAT = 13.80
Avg WT  = 8.20
```

SRTF (Preemtive)

```
SRTF (Preemptive) Gantt:
| P1 | P2 | P2 | P2 | P2 | P5 | P5 | P4 | P4 | P4 | P4 | P4 | P1 | P1 | P1 | P1 | P1 |
P1 | P1 | P3 | P3 | P3 | P3 | P3 | P3 | P3 | P3 | P3 |

PID Arrival   Burst     CT   TAT  WT
1    0    8    19   19   11
2    1    4    5    4    0
3    2    9    28   26   17
4    3    5    12   9    4
5    4    2    7    3    1
Avg TAT = 12.20
Avg WT  = 6.60
```

HRNN (Non-Preemtive)

```
WAS (Preemptive) Gantt (K=0.50):
| P1 (1) | P2 (5) | P5 (7) | P4 (12) | P1 (19) | P3 (28)|

PID Arrival   Burst     CT   TAT  WT
-------------------------------------------------
P1   0    8    19   19   11
P2   1    4    5    4    0
P3   2    9    28   26   17
P4   3    5    12   9    4
P5   4    2    7    3    1
-------------------------------------------------
Avg TAT = 12.20
Avg WT  = 6.60
```

WAS (Preemtive)

```
WAS (Preemptive) Gantt (K=0.50):
| P1 | P2 | P2 | P2 | P2 | P5 | P5 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P4 | P4 | P4 |
P4 | P4 | P3 | P3 | P3 | P3 | P3 | P3 | P3 | P3 | P3 | idle |

PID  Arrival   Burst     CT   TAT  WT
1    0    8    14   14   6
2    1    4    5    4    0
3    2    9    28   26   17
4    3    5    19   16   11
5    4    2    7    3    1
Avg TAT = 12.60
Avg WT  = 7.00
```

Performance Comparison Table

Algorithm	Average TAT	Average WT	Starvation	Preemptive
SJF	13.8	8.2	High	No
SRTF	12.2	6.6	Very High	Yes
HRRN	13.8	8.2	None	No
WAS	13.2	7.6	None	Yes

CONCLUSION

The Weighted Aging Shortest (WAS) algorithm successfully addresses the shortcomings of existing scheduling algorithms by introducing a balanced, tunable, and starvation-free scheduling mechanism. The results show that WAS consistently provides better fairness than SJF and SRTF, while maintaining strong performance characteristics. Its preemptive nature makes it suitable for responsive systems, and its simplicity ensures suitability for educational and academic environments.

This project demonstrates that hybrid scheduling techniques combining aging and burst-based metrics are highly effective for modern workloads. Future enhancements may include adaptive tuning of k, multi-core extensions, and integration into Linux kernel simulations.