**Operating System Project**

**Process Simulator:**Round Robin Scheduling

First come first serve & Shortest Job first Scheduling

Priority Scheduling

These three Algorithms are implemented with threads in Ubuntu operating system.  
The detail description and code are given below.  
  
**“How to run project in your System”**

Step 1: Open Ubuntu operating system.  
Step 2: Open any browser i.e. chrome or Firefox.  
Step 3: Copy my Repository link and open in your browser.  
Step 4: Download the zip file in any folder.  
Step 5: Extract all files and open terminal in same folder path.

Step 6: run command ‘ bash OS\_project.sh ’  
Step 7: Test all 4 algorithms  
Step 8: Select 5th option to stop execution.

Table of Contents

[1. INTRODUCTION: 3](#_Toc168840170)

[2. Problem Statement: 3](#_Toc168840171)

[3. Technique: 3](#_Toc168840172)

[4. Operating System: 4](#_Toc168840173)

[5. Languages: 4](#_Toc168840174)

[6. Implementation: 4](#_Toc168840175)

[7. Modules: 4](#_Toc168840176)

[Conclusion: 5](#_Toc168840177)

# **INTRODUCTION:**

The operating system plays a crucial role in managing computer resources and providing an efficient platform for executing various applications. One critical aspect of an operating system is its process scheduling algorithm, which determines the order in which processes are executed on the CPU. The choice of the scheduling algorithm significantly impacts the system's performance, response time, and overall efficiency.

In this project, our objective was to design and implement a new scheduling algorithm for an operating system. The motivation behind this project stems from the need to improve existing scheduling algorithms and to explore innovative approaches that can further optimize resource allocation and enhance overall system performance

# **Problem Statement:**

The problem we aimed to address in this project was the inefficiency and limitations of existing scheduling algorithms. Traditional scheduling algorithms, such as First-Come-First Serve (FCFS) and Round Robin (RR), are widely used but may not always provide the best performance in various scenarios.

Common issues with existing scheduling algorithms include high average waiting times, poor utilization of system resources, and the inability to prioritize certain processes based on their characteristics or priorities. These limitations can lead to reduced system performance, increased response times, and an overall degraded user experience.

To overcome these challenges, our goal was to develop a new scheduling algorithm that could optimize resource allocation, reduce waiting times, and improve overall system efficiency. The algorithm should consider factors such as process characteristics, priority levels, and resource utilization to make informed scheduling decisions and enhance the overall performance of the operating system.

# **Technique:**

We employed VMware as our virtualization platform to create and manage virtual machines. VMware allowed us to simulate multiple instances of the operating system environment on a single physical machine, enabling us to perform testing, debugging, and evaluation in a controlled and isolated setting. The use of virtualization technology provided flexibility and ease of deployment for our project.

# **Operating System:**

We selected Ubuntu as the base operating system for our project. Ubuntu is a popular and widely used Linux distribution that offers a robust and stable environment for software development. Its open-source nature, extensive community support, and compatibility with various software packages made it an ideal choice for our project. Additionally, Ubuntu provided the necessary tools and libraries required for implementing our custom scheduling algorithm

# **Languages:**

We primarily used the C programming language for the implementation of our operating system and the scheduling algorithm. C is a low-level programming language that offers direct access to system resources, making it well-suited for system-level programming. Its efficiency, performance, and close integration with the underlying hardware were key factors in selecting C as our language of choice.

# **Implementation:**

Using the chosen technologies and programming languages, we began implementing the components of the operating system. We followed a top-down approach, starting with the core functionalities and gradually adding more features. Throughout the implementation phase, we adhered to coding standards and practices to enhance code readability, maintainability, and reusability. We also utilized version control systems, such as Git, to track changes and collaborate effectively within the development team.

# **Modules:**

Round Robin Scheduling

FCFS & Shortest Job first Scheduling

Priority Scheduling

# **Conclusion:**

In conclusion, our project compared different scheduling algorithms in operating systems. We implemented and evaluated First-Come-First Serve (FCFS), Round Robin (RR), Shortest Job Next (SJN), and our newly developed algorithm. Our comparative analysis showed that the traditional FCFS algorithm leads to inefficiencies, while RR provides fair scheduling but may not be optimal for resource allocation. SJN improved performance but could suffer from starvation. Our new algorithm demonstrated superior results, reducing waiting times and optimizing resource utilization. Further research is needed to validate its performance in different environments. Overall, our project contributes to the optimization of operating systems and provides insights for future research in this area.

**Code:**

**SJF Scheduling with Threads**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

typedef struct {

int id; int burst\_time; int waiting\_time, int turnaround\_time;}

Process;

typedef struct {

int id; int burst\_time; int waiting\_time; int turnaround\_time;}

ThreadArgs;

int compare(const void \*a, const void \*b) {

Process \*processA = (Process \*)a;

Process \*processB = (Process \*)b;

return processA->burst\_time - processB->burst\_time;}

void\* execute\_process(void \*args) {

ThreadArgs \*process\_args = (ThreadArgs \*)args;

printf("Process %d with burst time %d is starting.\n", process\_args->id, process\_args->burst\_time);

sleep(process\_args->burst\_time);

printf("Process %d with burst time %d has finished.\n", process\_args->id, process\_args->burst\_time);

return NULL;}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n); Process processes[n]; pthread\_t threads[n];

ThreadArgs thread\_args[n];

for (int i = 0; i < n; i++) {

printf("Enter Burst/Service time for process %d: ", i);

scanf("%d", &processes[i].burst\_time);

processes[i].id = i; }

qsort(processes, n, sizeof(Process), compare);

processes[0].waiting\_time = 0;

processes[0].turnaround\_time = processes[0].burst\_time;

float total\_wt = 0, total\_tat = processes[0].turnaround\_time;

for (int i = 1; i < n; i++) {

processes[i].waiting\_time = processes[i-1].waiting\_time + processes[i-1].burst\_time;

processes[i].turnaround\_time = processes[i].waiting\_time + processes[i].burst\_time;

total\_wt += processes[i].waiting\_time;

total\_tat += processes[i].turnaround\_time; }

for (int i = 0; i < n; i++) {

thread\_args[i].id = processes[i].id;

thread\_args[i].burst\_time = processes[i].burst\_time;

thread\_args[i].waiting\_time = processes[i].waiting\_time;

thread\_args[i].turnaround\_time = processes[i].turnaround\_time;

pthread\_create(&threads[i], NULL, execute\_process, &thread\_args[i]);}

for (int i = 0; i < n; i++) {

pthread\_join(threads[i], NULL);}

printf("\n PROCESS \t BURST TIME \t WAITING TIME \t TURNAROUND TIME\n");

for (int i = 0; i < n; i++) {

printf("P%d \t\t %d \t\t %d \t\t %d\n", processes[i].id, processes[i].burst\_time, processes[i].waiting\_time, processes[i].turnaround\_time);}

printf("\nAverage waiting time: %f ", total\_wt/n);

printf("\nAverage turnaround time: %f \n", total\_tat/n);

return 0;}

**FCFS Scheduling with Threads**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

typedef struct {int id; int arrival\_time; int burst\_time; int waiting\_time; int finish\_time; int turnaround\_time;}

Process;

typedef struct { int id; int burst\_time; int waiting\_time; int finish\_time;}

ThreadArgs;

void\* execute\_process(void \*args) {

ThreadArgs \*process\_args = (ThreadArgs \*)args;

printf("Process %d with burst time %d is starting.\n", process\_args->id, process\_args->burst\_time);

sleep(process\_args->burst\_time);

printf("Process %d with burst time %d has finished.\n", process\_args->id, process\_args->burst\_time);

pthread\_exit(0);}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

Process processes[n];

pthread\_t threads[n];

ThreadArgs thread\_args[n];

for (int i = 0; i < n; i++) {

printf("Enter Arrival time for process %d: ", i);

scanf("%d", &processes[i].arrival\_time);

printf("Enter Burst/Service time for process %d: ", i);

scanf("%d", &processes[i].burst\_time);

processes[i].id = i; } processes[0].waiting\_time = 0; processes[0].finish\_time = processes[0].burst\_time;

processes[0].turnaround\_time = processes[0].burst\_time;

float total\_wt = 0, total\_tat = processes[0].turnaround\_time;

for (int i = 1; i < n; i++) {

processes[i].waiting\_time = processes[i-1].finish\_time - processes[i].arrival\_time;

if (processes[i].waiting\_time < 0) {

processes[i].waiting\_time = 0;}

processes[i].finish\_time = processes[i].waiting\_time + processes[i].burst\_time + processes[i].arrival\_time;

processes[i].turnaround\_time = processes[i].finish\_time - processes[i].arrival\_time;

total\_wt += processes[i].waiting\_time;

total\_tat += processes[i].turnaround\_time;}

for (int i = 0; i < n; i++) {

thread\_args[i].id = processes[i].id;

thread\_args[i].burst\_time = processes[i].burst\_time;

thread\_args[i].waiting\_time = processes[i].waiting\_time;

thread\_args[i].finish\_time = processes[i].finish\_time;

pthread\_create(&threads[i], NULL, execute\_process, &thread\_args[i]);

pthread\_join(threads[i], NULL); // Ensuring FCFS order }

printf("\n PROCESS \t ARRIVAL TIME \t SERVICE TIME \t WAITING TIME \t FINISH TIME \t TURNAROUND TIME\n");

for (int i = 0; i < n; i++) {

printf("P%d \t\t %d \t\t %d \t\t %d \t\t %d \t\t %d\n", processes[i].id, processes[i].arrival\_time, processes[i].burst\_time, processes[i].waiting\_time, processes[i].finish\_time, processes[i].turnaround\_time); }

printf("\nAverage waiting time: %f ", total\_wt/n);

printf("\nAverage turnaround time: %f \n", total\_tat/n);

return 0; }

**Priority Scheduling with Threads**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

typedef struct { int id; int burst\_time; int priority; int waiting\_time; int turnaround\_time;}

Process;

typedef struct { int id; int burst\_time;}

ThreadArgs;

int compare(const void \*a, const void \*b) {

Process \*processA = (Process \*)a;

Process \*processB = (Process \*)b;

return processA->priority - processB->priority;}

void\* execute\_process(void \*args) {

ThreadArgs \*process\_args = (ThreadArgs \*)args;

printf("Process %d with burst time %d is starting.\n", process\_args->id, process\_args->burst\_time);

sleep(process\_args->burst\_time);

printf("Process %d with burst time %d has finished.\n", process\_args->id, process\_args->burst\_time); return NULL;}

int main() { int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

Process processes[n]; pthread\_t threads[n];

ThreadArgs thread\_args[n];

for (int i = 0; i < n; i++) {

printf("Enter Burst/Service time for process %d: ", i);

scanf("%d", &processes[i].burst\_time);

printf("Enter Priority for process %d (lower number indicates higher priority): ", i);

scanf("%d", &processes[i].priority);

processes[i].id = i; }

qsort(processes, n, sizeof(Process), compare);

processes[0].waiting\_time = 0;

processes[0].turnaround\_time = processes[0].burst\_time;

float total\_wt = 0, total\_tat = processes[0].turnaround\_time;

for (int i = 1; i < n; i++) {

processes[i].waiting\_time = processes[i-1].waiting\_time + processes[i-1].burst\_time;

processes[i].turnaround\_time = processes[i].waiting\_time + processes[i].burst\_time;

total\_wt += processes[i].waiting\_time;

total\_tat += processes[i].turnaround\_time;}

for (int i = 0; i < n; i++) {

thread\_args[i].id = processes[i].id;

thread\_args[i].burst\_time = processes[i].burst\_time;

pthread\_create(&threads[i], NULL, execute\_process, &thread\_args[i]);

pthread\_join(threads[i], NULL); }

printf("\n PROCESS \t PRIORITY \t BURST TIME \t WAITING TIME \t TURNAROUND TIME\n");

for (int i = 0; i < n; i++) {

printf("P%d \t\t %d \t\t %d \t\t %d \t\t %d\n", processes[i].id, processes[i].priority, processes[i].burst\_time, processes[i].waiting\_time, processes[i].turnaround\_time); }

printf("\nAverage waiting time: %f ", total\_wt/n);

printf("\nAverage turnaround time: %f \n", total\_tat/n);

return 0;}

**Round Robin Scheduling with Threads =**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

typedef struct { int id; int burst\_time; int remaining\_time; int arrival\_time; int waiting\_time; int turnaround\_time;} Process;

typedef struct { int id; int quantum; int \*remaining\_time;} ThreadArgs;

pthread\_mutex\_t lock;

int current\_time = 0;

void\* execute\_process(void \*args) {

ThreadArgs \*process\_args = (ThreadArgs \*)args;

while (\*process\_args->remaining\_time > 0) {

pthread\_mutex\_lock(&lock);

if (\*process\_args->remaining\_time > 0) {

int time\_slice = (\*process\_args->remaining\_time > process\_args->quantum) ? process\_args->quantum : \*process\_args->remaining\_time;

printf("Process %d is executing for %d time units.\n", process\_args->id, time\_slice);

sleep(time\_slice);

current\_time += time\_slice;

\*process\_args->remaining\_time -= time\_slice;

printf("Process %d has %d time units remaining.\n", process\_args->id, \*process\_args->remaining\_time);}

pthread\_mutex\_unlock(&lock);

sleep(1); }

return NULL; }

int main() {

int n, quantum;

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter the quantum time: ");

scanf("%d", &quantum);

Process processes[n];

pthread\_t threads[n];

ThreadArgs thread\_args[n];

for (int i = 0; i < n; i++) {

printf("Enter Burst/Service time for process %d: ", i);

scanf("%d", &processes[i].burst\_time);

processes[i].id = i;

processes[i].remaining\_time = processes[i].burst\_time;

processes[i].waiting\_time = 0;

processes[i].turnaround\_time = 0; }

pthread\_mutex\_init(&lock, NULL);

for (int i = 0; i < n; i++) {

thread\_args[i].id = processes[i].id;

thread\_args[i].quantum = quantum;

thread\_args[i].remaining\_time = &processes[i].remaining\_time;

pthread\_create(&threads[i], NULL, execute\_process, &thread\_args[i]); }

for (int i = 0; i < n; i++) {

pthread\_join(threads[i], NULL); }

for (int i = 0; i < n; i++) {

processes[i].turnaround\_time = current\_time - processes[i].arrival\_time;

processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time; } float total\_wt = 0, total\_tat = 0;

printf("\n PROCESS \t BURST TIME \t WAITING TIME \t TURNAROUND TIME\n");

for (int i = 0; i < n; i++) {

printf("P%d \t\t %d \t\t %d \t\t %d\n", processes[i].id, processes[i].burst\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

total\_wt += processes[i].waiting\_time;

total\_tat += processes[i].turnaround\_time;

}

printf("\nAverage waiting time: %f ", total\_wt/n);

printf("\nAverage turnaround time: %f \n", total\_tat/n);

pthread\_mutex\_destroy(&lock); return 0;





