1. **FCFS:**

CODE:

#include <stdio.h>

int main() {

int atchoice, at[100], bt[100], wt[100], tat[100], pid;

float total\_wt = 0, total\_tat = 0;

printf("FCFS Algorithm\n");

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

scanf("%d", &pid);

printf("Enter the burst time of each process: ");

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

printf("\nProcess %d: ", i + 1);

scanf("%d", &bt[i]);

}

printf("Enter the choice with arrival time(0) or without arrival time(1): ");

scanf("%d", &atchoice);

if (atchoice == 0) {

printf("Implementing with arrival time\n");

printf("Enter the arrival time of each process: ");

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

printf("\nProcess %d: ", i + 1);

scanf("%d", &at[i]);

}

} else if (atchoice == 1) {

printf("Implementing without arrival time\n");

} else {

printf("Invalid Input!\n");

return 1;

}

// Calculate waiting time and turnaround time

wt[0] = 0; // First process has 0 waiting time

tat[0] = bt[0]; // First process's turnaround time is its burst time

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

wt[i] = wt[i - 1] + bt[i - 1]; // Waiting time is the sum of burst times of previous processes

tat[i] = wt[i] + bt[i]; // Turnaround time is waiting time plus burst time

}

// Display waiting time and turnaround time for each process

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("P%d\t\t%d\t\t\t\t%d\t\t\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);

total\_wt += wt[i];

total\_tat += tat[i];

}

float average\_wt = total\_wt / pid; // Calculate average waiting time

float average\_tat = total\_tat / pid; // Calculate average turnaround time

printf("\nAverage Waiting Time: %.2f\n", average\_wt);

printf("Average Turnaround Time: %.2f\n", average\_tat);

return 0;

}

1. **SJF (Pre emptive):**

CODE:

#include <stdio.h>

// Structure to represent a process

struct Process {

int pid; // Process ID

int burstTime; // Burst time

int arrivalTime; // Arrival time

int remainingTime; // Remaining burst time

};

int main() {

int atchoice, pid;

float total\_wt = 0, total\_tat = 0;

printf("SJF (Preemptive) Algorithm\n");

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

scanf("%d", &pid);

// Declare an array of struct Process

struct Process processes[pid];

// Input burst time and arrival time for each process

printf("Enter the burst time and arrival time of each process:\n");

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

printf("Process %d Burst Time: ", i + 1);

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

processes[i].remainingTime = processes[i].burstTime;

printf("Process %d Arrival Time: ", i + 1);

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

processes[i].pid = i + 1;

}

// Sort processes based on arrival time

for (int i = 0; i < pid - 1; i++) {

for (int j = 0; j < pid - i - 1; j++) {

if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {

// Swap processes

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

// Implement SJF (Preemptive) algorithm

int currentTime = processes[0].arrivalTime;

int completed = 0; // Counter to keep track of completed processes

printf("\n");

printf("---------------------------------------------------------------------\n");

printf("| Process | Burst Time | Arrival Time | Waiting Time | Turnaround Time |\n");

printf("---------------------------------------------------------------------\n");

while (completed < pid) {

int shortest = -1; // Index of the process with shortest remaining time

int shortestTime = 9999; // Set initial shortest time to a large value

// Find the process with the shortest remaining time among arrived processes

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

if (processes[i].arrivalTime <= currentTime && processes[i].remainingTime < shortestTime && processes[i].remainingTime > 0) {

shortest = i;

shortestTime = processes[i].remainingTime;

}

}

// If no process is found, increment current time

if (shortest == -1) {

currentTime++;

continue;

}

// Execute the shortest process for 1 unit

processes[shortest].remainingTime--;

currentTime++;

// If process is completed, calculate waiting and turnaround time

if (processes[shortest].remainingTime == 0) {

completed++;

int turnaroundTime = currentTime - processes[shortest].arrivalTime;

int waitingTime = turnaroundTime - processes[shortest].burstTime;

total\_wt += waitingTime;

total\_tat += turnaroundTime;

// Print process details

printf("| P%d | %d | %d | %d | %d |\n", processes[shortest].pid, processes[shortest].burstTime,

processes[shortest].arrivalTime, waitingTime, turnaroundTime);

}

}

printf("---------------------------------------------------------------------\n");

float average\_wt = total\_wt / pid;

float average\_tat = total\_tat / pid;

printf("\nAverage Waiting Time: %.2f\n", average\_wt);

printf("Average Turnaround Time: %.2f\n", average\_tat);

return 0;

}

1. **SJF (Non – Pre emptive):**

CODE:

#include <stdio.h>

// Structure to represent a process

struct Process {

int pid; // Process ID

int burstTime; // Burst time

int arrivalTime; // Arrival time

int remainingTime; // Remaining burst time

};

int main() {

int pid;

float total\_wt = 0, total\_tat = 0;

printf("SJF (Non-Preemptive) Algorithm\n");

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

scanf("%d", &pid);

// Declare an array of struct Process

struct Process processes[pid];

// Input burst time and arrival time for each process

printf("Enter the burst time and arrival time of each process:\n");

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

printf("Process %d Burst Time: ", i + 1);

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

processes[i].remainingTime = processes[i].burstTime;

printf("Process %d Arrival Time: ", i + 1);

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

processes[i].pid = i + 1;

}

// Sort processes based on burst time (SJF)

for (int i = 0; i < pid - 1; i++) {

for (int j = 0; j < pid - i - 1; j++) {

if (processes[j].burstTime > processes[j + 1].burstTime) {

// Swap processes

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

int currentTime = 0; // Start time from 0

printf("\n");

printf("---------------------------------------------------------------------\n");

printf("| Process | Burst Time | Arrival Time | Waiting Time | Turnaround Time |\n");

printf("---------------------------------------------------------------------\n");

// Calculate waiting time and turnaround time

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

int turnaroundTime = currentTime + processes[i].burstTime - processes[i].arrivalTime;

int waitingTime = currentTime - processes[i].arrivalTime;

if (waitingTime < 0) {

waitingTime = 0;

currentTime = processes[i].arrivalTime;

}

total\_wt += waitingTime;

total\_tat += turnaroundTime;

// Print process details

printf("| P%d | %d | %d | %d | %d |\n", processes[i].pid, processes[i].burstTime,

processes[i].arrivalTime, waitingTime, turnaroundTime);

// Update current time

currentTime += processes[i].burstTime;

}

printf("---------------------------------------------------------------------\n");

float average\_wt = total\_wt / pid;

float average\_tat = total\_tat / pid;

printf("\nAverage Waiting Time: %.2f\n", average\_wt);

printf("Average Turnaround Time: %.2f\n", average\_tat);

return 0;

}

1. **Priority (Highest Order Priority):**

CODE:

#include <stdio.h>

// Structure to represent a process

struct Process {

int pid; // Process ID

int burstTime; // Burst time

int arrivalTime; // Arrival time

int priority; // Priority

int remainingTime; // Remaining burst time

};

int main() {

int pid;

float total\_wt = 0, total\_tat = 0;

printf("Priority (Highest Priority First) Algorithm\n");

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

scanf("%d", &pid);

// Declare an array of struct Process

struct Process processes[pid];

// Input burst time, arrival time, and priority for each process

printf("Enter the burst time, arrival time, and priority of each process:\n");

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

printf("Process %d Burst Time: ", i + 1);

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

processes[i].remainingTime = processes[i].burstTime;

printf("Process %d Arrival Time: ", i + 1);

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

printf("Process %d Priority: ", i + 1);

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

processes[i].pid = i + 1;

}

// Sort processes based on priority (Highest Priority First)

for (int i = 0; i < pid - 1; i++) {

for (int j = 0; j < pid - i - 1; j++) {

if (processes[j].priority < processes[j + 1].priority) {

// Swap processes

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

int currentTime = 0; // Start time from 0

printf("\n");

printf("---------------------------------------------------------------------\n");

printf("| Process | Burst Time | Arrival Time | Waiting Time | Turnaround Time |\n");

printf("---------------------------------------------------------------------\n");

// Calculate waiting time and turnaround time

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

int turnaroundTime = currentTime + processes[i].burstTime - processes[i].arrivalTime;

int waitingTime = currentTime - processes[i].arrivalTime;

if (waitingTime < 0) {

waitingTime = 0;

currentTime = processes[i].arrivalTime;

}

total\_wt += waitingTime;

total\_tat += turnaroundTime;

// Print process details

printf("| P%d | %d | %d | %d | %d |\n", processes[i].pid, processes[i].burstTime,

processes[i].arrivalTime, waitingTime, turnaroundTime);

// Update current time

currentTime += processes[i].burstTime;

}

printf("---------------------------------------------------------------------\n");

float average\_wt = total\_wt / pid;

float average\_tat = total\_tat / pid;

printf("\nAverage Waiting Time: %.2f\n", average\_wt);

printf("Average Turnaround Time: %.2f\n", average\_tat);

return 0;

}

1. **Priority (Least Order Priority):**

CODE:

#include <stdio.h>

// Structure to represent a process

struct Process {

int pid; // Process ID

int burstTime; // Burst time

int arrivalTime; // Arrival time

int priority; // Priority

int remainingTime; // Remaining burst time

};

int main() {

int pid;

float total\_wt = 0, total\_tat = 0;

printf("Priority (Least Priority First) Algorithm\n");

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

scanf("%d", &pid);

// Declare an array of struct Process

struct Process processes[pid];

// Input burst time, arrival time, and priority for each process

printf("Enter the burst time, arrival time, and priority of each process:\n");

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

printf("Process %d Burst Time: ", i + 1);

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

processes[i].remainingTime = processes[i].burstTime;

printf("Process %d Arrival Time: ", i + 1);

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

printf("Process %d Priority: ", i + 1);

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

processes[i].pid = i + 1;

}

// Sort processes based on priority (Least Priority First)

for (int i = 0; i < pid - 1; i++) {

for (int j = 0; j < pid - i - 1; j++) {

if (processes[j].priority > processes[j + 1].priority) {

// Swap processes

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

int currentTime = 0; // Start time from 0

printf("\n");

printf("---------------------------------------------------------------------\n");

printf("| Process | Burst Time | Arrival Time | Waiting Time | Turnaround Time |\n");

printf("---------------------------------------------------------------------\n");

// Calculate waiting time and turnaround time

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

int turnaroundTime = currentTime + processes[i].burstTime - processes[i].arrivalTime;

int waitingTime = currentTime - processes[i].arrivalTime;

if (waitingTime < 0) {

waitingTime = 0;

currentTime = processes[i].arrivalTime;

}

total\_wt += waitingTime;

total\_tat += turnaroundTime;

// Print process details

printf("| P%d | %d | %d | %d | %d |\n", processes[i].pid, processes[i].burstTime,

processes[i].arrivalTime, waitingTime, turnaroundTime);

// Update current time

currentTime += processes[i].burstTime;

}

printf("---------------------------------------------------------------------\n");

float average\_wt = total\_wt / pid;

float average\_tat = total\_tat / pid;

printf("\nAverage Waiting Time: %.2f\n", average\_wt);

printf("Average Turnaround Time: %.2f\n", average\_tat);

return 0;

}

1. **Round Robin:**

CODE:

#include <stdio.h>

// Structure to represent a process

struct Process {

int pid; // Process ID

int burstTime; // Burst time

int arrivalTime; // Arrival time

int remainingTime; // Remaining burst time

};

int main() {

int pid, timeQuantum;

float total\_wt = 0, total\_tat = 0;

printf("Round Robin Algorithm\n");

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

scanf("%d", &pid);

printf("Enter the time quantum: ");

scanf("%d", &timeQuantum);

// Declare an array of struct Process

struct Process processes[pid];

// Input burst time and arrival time for each process

printf("Enter the burst time and arrival time of each process:\n");

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

printf("Process %d Burst Time: ", i + 1);

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

processes[i].remainingTime = processes[i].burstTime;

printf("Process %d Arrival Time: ", i + 1);

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

processes[i].pid = i + 1;

}

int currentTime = 0; // Start time from 0

int completed = 0; // Counter to keep track of completed processes

printf("\n");

printf("---------------------------------------------------------------------\n");

printf("| Process | Burst Time | Arrival Time | Waiting Time | Turnaround Time |\n");

printf("---------------------------------------------------------------------\n");

while (completed < pid) {

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

if (processes[i].remainingTime > 0) {

int executeTime = (processes[i].remainingTime > timeQuantum) ? timeQuantum : processes[i].remainingTime;

processes[i].remainingTime -= executeTime;

currentTime += executeTime;

// If process is completed, calculate waiting and turnaround time

if (processes[i].remainingTime == 0) {

completed++;

int turnaroundTime = currentTime - processes[i].arrivalTime;

int waitingTime = turnaroundTime - processes[i].burstTime;

total\_wt += waitingTime;

total\_tat += turnaroundTime;

// Print process details

printf("| P%d | %d | %d | %d | %d |\n", processes[i].pid, processes[i].burstTime,

processes[i].arrivalTime, waitingTime, turnaroundTime);

}

}

}

}

printf("---------------------------------------------------------------------\n");

float average\_wt = total\_wt / pid;

float average\_tat = total\_tat / pid;

printf("\nAverage Waiting Time: %.2f\n", average\_wt);

printf("Average Turnaround Time: %.2f\n", average\_tat);

return 0;

}