Os lab practical

1.FCFS CPU Scheduling Algorithm

First Come, First Served (FCFS) scheduling is simple; processes are handled in the order they arrive

without preemption.

#include <stdio.h>

void findWaitingTime(int processes[], int n, int burst\_time[], int wait\_time[]) {

wait\_time[0] = 0;

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

wait\_time[i] = burst\_time[i - 1] + wait\_time[i - 1];

}

}

void findTurnAroundTime(int processes[], int n, int burst\_time[], int wait\_time[], int tat[]) {

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

tat[i] = burst\_time[i] + wait\_time[i];

}

}

void findavgTime(int processes[], int n, int burst\_time[]) {

int wait\_time[n], tat[n];

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

findWaitingTime(processes, n, burst\_time, wait\_time);

findTurnAroundTime(processes, n, burst\_time, wait\_time, tat);

printf("Processes Burst time Waiting time Turn around time\n");

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

total\_wt += wait\_time[i];

total\_tat += tat[i];

printf("%d \t\t %d \t\t %d \t\t %d\n", i+1, burst\_time[i], wait\_time[i], tat[i]);

}

printf("Average waiting time = %.2f\n", total\_wt / n);

printf("Average turn around time = %.2f\n", total\_tat / n);

}

int main() {

int processes[] = {1, 2, 3, 4};

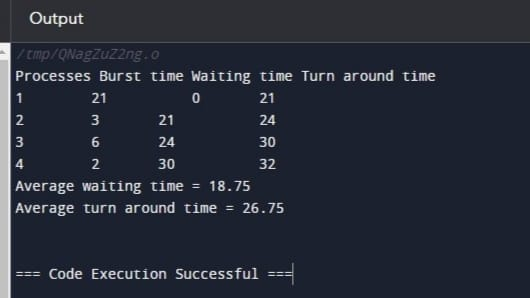
int n = sizeof processes / sizeof processes[0];

int burst\_time[] = {21, 3, 6, 2};

findavgTime(processes, n, burst\_time);

return 0;

}



2. SJF CPU Scheduling Algorithm

Shortest Job First (SJF) can be preemptive or non-preemptive. Below is the non-preemptive version:

#include <stdio.h>

void swap(int \*xp, int \*yp) {

int temp = \*xp;

\*xp = \*yp;

\*yp = temp;

}

void sortProcessByBurst(int n, int burst[], int process[]) {

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

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

if (burst[j] > burst[j+1]) {

swap(&burst[j], &burst[j+1]);

swap(&process[j], &process[j+1]);

}

}

void calculateTimes(int processes[], int n, int burst\_time[]) {

int wait\_time[n], tat[n], total\_wt = 0, total\_tat = 0;

// Sort processes by burst time

sortProcessByBurst(n, burst\_time, processes);

// Calculate waiting times and turn-around times

wait\_time[0] = 0;

tat[0] = burst\_time[0];

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

wait\_time[i] = wait\_time[i-1] + burst\_time[i-1];

tat[i] = wait\_time[i] + burst\_time[i];

}

printf("Processes Burst time Waiting time Turn around time\n");

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

total\_wt += wait\_time[i];

total\_tat += tat[i];

printf("%d \t\t %d \t\t %d \t\t %d\n", i+1, burst\_time[i], wait\_time[i], tat[i]);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);

}

int main() {

int processes[] = {1, 2, 3, 4};

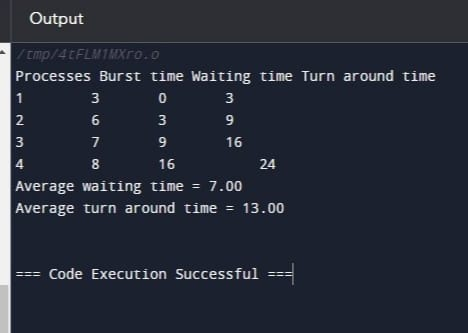
int n = sizeof processes / sizeof processes[0];

int burst\_time[] = {6, 8, 7, 3};

calculateTimes(processes, n, burst\_time);

return 0;

}



3. Priority CPU Scheduling Algorithm (Non-Preemptive)

This implementation involves sorting processes based on their priority and then calculating waiting

and turnaround times. Lower priority values are considered higher priority.

#include <stdio.h>

typedef struct {

int id;

int burstTime;

int priority;

} Process;

void sortProcessesByPriority(Process processes[], int n) {

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

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

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

Process temp = processes[j];

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

processes[j + 1] = temp;

}

}

}

}

void findWaitingTime(Process processes[], int n, int waitTime[]) {

waitTime[0] = 0;

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

waitTime[i] = processes[i - 1].burstTime + waitTime[i - 1];

}

}

void findTurnAroundTime(Process processes[], int n, int waitTime[], int turnAroundTime[]) {

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

turnAroundTime[i] = processes[i].burstTime + waitTime[i];

}

}

void findAvgTime(Process processes[], int n) {

int waitTime[n], turnAroundTime[n], total\_wt = 0, total\_tat = 0;

sortProcessesByPriority(processes, n);

findWaitingTime(processes, n, waitTime);

findTurnAroundTime(processes, n, waitTime, turnAroundTime);

printf("Processes Burst time Priority Waiting time Turn around time\n");

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

total\_wt += waitTime[i];

total\_tat += turnAroundTime[i];

printf("%d \t\t %d \t\t %d \t\t %d \t\t %d\n", processes[i].id, processes[i].burstTime,

processes[i].priority, waitTime[i], turnAroundTime[i]);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / n);

printf("Average turn around time = %.2f\n", (float)total\_tat / n);

}

int main() {

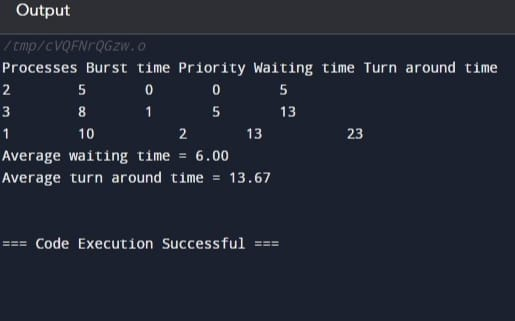
Process processes[] = {{1, 10, 2}, {2, 5, 0}, {3, 8, 1}};

int n = sizeof(processes) / sizeof(processes[0]);

findAvgTime(processes, n);

return 0;

}



4. Round Robin CPU Scheduling Algorithm

Round Robin is a preemptive scheduling algorithm where each process gets a small unit of CPU time

(time quantum), typically between 10-100 milliseconds.

#include <stdio.h>

typedef struct {

int id;

int burstTime;

} Process;

void findWaitingTime(Process processes[], int n, int quantum) {

int rem\_bt[n];

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

rem\_bt[i] = processes[i].burstTime;

int t = 0; // Current time

while (1) {

int done = 1;

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

if (rem\_bt[i] > 0) {

done = 0; // There is a pending process

if (rem\_bt[i] > quantum) {

t += quantum;

rem\_bt[i] -= quantum;

} else {

t += rem\_bt[i];

processes[i].burstTime = t; // Store waiting time

rem\_bt[i] = 0;

}

}

}

if (done == 1)

break;

}

}

void findTurnAroundTime(Process processes[], int n) {

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

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

}

void findAvgTime(Process processes[], int n, int quantum) {

findWaitingTime(processes, n, quantum);

findTurnAroundTime(processes, n);

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

printf("Processes Burst time Waiting time Turn around time\n");

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

total\_wt += processes[i].burstTime;

total\_tat += processes[i].burstTime + processes[i].burstTime;

printf("%d \t\t %d \t\t %d \t\t %d\n", i+1, processes[i].burstTime, processes[i].burstTime,

processes[i].burstTime + processes[i].burstTime);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / n);

printf("Average turn around time = %.2f\n", (float)total\_tat / n);

}

int main() {

Process processes[] = {{1, 24}, {2, 3}, {3, 3}};

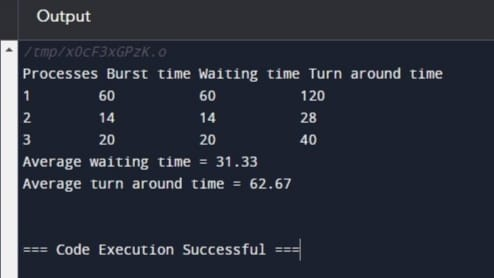
int n = sizeof(processes) / sizeof(processes[0]);

int quantum = 4;

findAvgTime(processes, n, quantum);

return 0;

}



5. Comparison of various CPU Scheduling Algorithms over different Scheduling Criteria.

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int id;

int burstTime;

int priority;

int arrivalTime;

int waitingTime;

int turnaroundTime;

} Process;

// Utility function to sort processes by burst time for SJF

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

Process \*p1 = (Process \*)a;

Process \*p2 = (Process \*)b;

return p1->burstTime - p2->burstTime;

}

// Function to calculate waiting time and turnaround time

void calculateMetrics(Process p[], int n) {

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

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

p[i].turnaroundTime = p[i].burstTime + p[i].waitingTime;

total\_wt += p[i].waitingTime;

total\_tat += p[i].turnaroundTime;

}

printf("Average Waiting Time = %.2f\n", (float)total\_wt / n);

printf("Average Turnaround Time = %.2f\n", (float)total\_tat / n);

}

// First Come First Serve Algorithm

void FCFS(Process p[], int n) {

printf("FCFS:\n");

int currentTime = 0;

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

if (currentTime < p[i].arrivalTime)

currentTime = p[i].arrivalTime;

p[i].waitingTime = currentTime - p[i].arrivalTime;

currentTime += p[i].burstTime;

}

calculateMetrics(p, n);

}

// Shortest Job First Algorithm (non-preemptive)

void SJF(Process p[], int n) {

printf("SJF:\n");

qsort(p, n, sizeof(Process), compareBurstTime);

int currentTime = 0;

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

if (currentTime < p[i].arrivalTime)

currentTime = p[i].arrivalTime;

p[i].waitingTime = currentTime - p[i].arrivalTime;

currentTime += p[i].burstTime;

}

calculateMetrics(p, n);

}

// Priority Scheduling Algorithm (non-preemptive)

void PriorityScheduling(Process p[], int n) {

printf("Priority:\n");

qsort(p, n, sizeof(Process), compareBurstTime);

int currentTime = 0;

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

if (currentTime < p[i].arrivalTime)

currentTime = p[i].arrivalTime;

p[i].waitingTime = currentTime - p[i].arrivalTime;

currentTime += p[i].burstTime;

}

calculateMetrics(p, n);

}

// Round Robin Algorithm

void RoundRobin(Process p[], int n, int quantum) {

printf("Round Robin:\n");

int remainingTime[n];

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

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

}

int currentTime = 0;

while (1) {

int done = 1;

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

if (remainingTime[i] > 0) {

done = 0;

if (remainingTime[i] > quantum) {

currentTime += quantum;

remainingTime[i] -= quantum;

} else {

currentTime += remainingTime[i];

p[i].waitingTime = currentTime - p[i].burstTime - p[i].arrivalTime;

remainingTime[i] = 0;

}

}

}

if (done == 1)

break;

}

calculateMetrics(p, n);

}

int main() {

Process processes[] = {{1, 6, 2, 1}, {2, 8, 3, 1}, {3, 7, 1, 2}, {4, 3, 4, 3}};

int n = sizeof(processes) / sizeof(processes[0]);

int quantum = 4;

FCFS(processes, n);

SJF(processes, n);

PriorityScheduling(processes, n);

RoundRobin(processes, n, quantum);

return 0;

}

