**Assignment 2**

**Set A**

**1. Write the simulation program using FCFS. The arrival time and first CPU bursts of**

**different jobs should be input to the system. Assume the fixed I/O waiting time (2**

**units). The next CPU burstshould be generated using random function. The output**

**should give the Gantt chart, Turnaround Time and Waiting time for each process and**

**average times.**

**$ touch fcfs3.c**

**$ gcc fcfs3.c -o fcfs3**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define IO\_WAIT 2

void generateRandomBurst(int \*burst\_time) {

\*burst\_time = (rand() % 10) + 1; // Generate a random CPU burst time between 1 and 10

}

void findWaitingTime(int arrival\_time[], int burst\_time[], int waiting\_time[], int n) {

int completion\_time[n];

int current\_time = 0;

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

if (current\_time < arrival\_time[i]) {

current\_time = arrival\_time[i];

}

waiting\_time[i] = current\_time - arrival\_time[i];

current\_time += burst\_time[i] + IO\_WAIT; // Add burst time and I/O wait

completion\_time[i] = current\_time;

}

}

void findTurnAroundTime(int burst\_time[], int waiting\_time[], int turnaround\_time[], int n) {

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

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

}

}

void printGanttChart(int arrival\_time[], int burst\_time[], int n) {

int completion\_time[n];

int current\_time = 0;

printf("Gantt Chart:\n");

printf("|");

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

if (current\_time < arrival\_time[i]) {

current\_time = arrival\_time[i];

}

printf(" P%d |", i + 1);

current\_time += burst\_time[i] + IO\_WAIT; // Add burst time and I/O wait

completion\_time[i] = current\_time;

}

printf("\n");

printf("0");

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

printf(" %2d", completion\_time[i]);

}

printf("\n");

}

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

int waiting\_time[n], turnaround\_time[n];

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

findWaitingTime(arrival\_time, burst\_time, waiting\_time, n);

findTurnAroundTime(burst\_time, waiting\_time, turnaround\_time, n);

printGanttChart(arrival\_time, burst\_time, n);

printf("Processes Burst time Arrival time Waiting time Turnaround time\n");

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

total\_waiting\_time += waiting\_time[i];

total\_turnaround\_time += turnaround\_time[i];

printf(" %d %d %d %d %d\n",

i + 1, burst\_time[i], arrival\_time[i], waiting\_time[i], turnaround\_time[i]);

}

float avg\_waiting\_time = (float)total\_waiting\_time / n;

float avg\_turnaround\_time = (float)total\_turnaround\_time / n;

printf("Average waiting time = %.2f\n", avg\_waiting\_time);

printf("Average turnaround time = %.2f\n", avg\_turnaround\_time);

}

int main() {

srand(time(NULL)); // Seed the random number generator

int n;

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

scanf("%d", &n);

int arrival\_time[n], burst\_time[n];

// Input arrival times and initial burst times

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

printf("Enter arrival time for process %d: ", i + 1);

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

printf("Enter first CPU burst time for process %d: ", i + 1);

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

}

// Generate next CPU bursts and update burst times

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

generateRandomBurst(&burst\_time[i]); // Generate a new burst time for the next cycle

}

// Calculate and display average waiting time and turnaround time

findavgTime(arrival\_time, burst\_time, n);

return 0;

}

**Output:**

**$ ./fcfs3**

Enter the number of processes: 4

Enter arrival time for process 1: 1

Enter first CPU burst time for process 1: 3

Enter arrival time for process 2: 3

Enter first CPU burst time for process 2: 4

Enter arrival time for process 3: 2

Enter first CPU burst time for process 3: 2

Enter arrival time for process 4: 4

Enter first CPU burst time for process 4: 6

Gantt Chart:

| P1 | P2 | P3 | P4 |

0 10 14 24 27

Processes Burst time Arrival time Waiting time Turnaround time

1 7 1 0 7

2 2 3 7 9

3 8 2 12 20

4 1 4 20 21

Average waiting time = 9.75

Average turnaround time = 14.25

**2. Write the simulation program using SJF(non-preemptive). The arrival time and first**

**CPU bursts of different jobs should be input to the system. The Assume the fixed I/O**

**waiting time (2 units).The next CPU burst should be generated using random**

**function. The output should give the Gantt chart, Turnaround Time and Waiting time**

**for each process and average times.**

**ANS:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

typedef struct {

char name[10];

int arrival\_time;

int cpu\_burst;

int waiting\_time;

int turnaround\_time;

} Job;

int generate\_next\_burst() {

return rand() % 10 + 1;

}

void sjf\_scheduling(Job jobs[], int num\_jobs) {

int time = 0, completed = 0;

bool finished[num\_jobs];

for (int i = 0; i < num\_jobs; i++) {

finished[i] = false;

}

while (completed < num\_jobs) {

int min\_burst = 1000;

int index = -1;

for (int i = 0; i < num\_jobs; i++) {

if (jobs[i].arrival\_time <= time && !finished[i] && jobs[i].cpu\_burst < min\_burst) {

min\_burst = jobs[i].cpu\_burst;

index = i;

}

}

if (index != -1) {

time += jobs[index].cpu\_burst + 2;

jobs[index].turnaround\_time = time - jobs[index].arrival\_time;

jobs[index].waiting\_time = jobs[index].turnaround\_time - jobs[index].cpu\_burst;

finished[index] = true;

completed++;

jobs[index].cpu\_burst = generate\_next\_burst();

} else {

time++;

}

}

}

int main() {

int num\_jobs;

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

scanf("%d", &num\_jobs);

Job jobs[num\_jobs];

for (int i = 0; i < num\_jobs; i++) {

printf("Enter arrival time for Job %d: ", i + 1);

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

printf("Enter first CPU burst for Job %d: ", i + 1);

scanf("%d", &jobs[i].cpu\_burst);

snprintf(jobs[i].name, sizeof(jobs[i].name), "Job %d", i + 1);

}

sjf\_scheduling(jobs, num\_jobs);

printf("\nJob\tTurnaround Time\tWaiting Time\n");

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

for (int i = 0; i < num\_jobs; i++) {

printf("%s\t%d\t\t%d\n", jobs[i].name, jobs[i].turnaround\_time, jobs[i].waiting\_time);

total\_turnaround\_time += jobs[i].turnaround\_time;

total\_waiting\_time += jobs[i].waiting\_time;

}

printf("\nAverage Turnaround Time: %.2f\n", (float)total\_turnaround\_time / num\_jobs);

printf("Average Waiting Time: %.2f\n", (float)total\_waiting\_time / num\_jobs);

return 0;

}

**gcc sjf\_scheduler.c -o sjf\_scheduler**

**./sjf\_scheduler**

**Output:**

Enter the number of jobs: 3

Enter arrival time for Job 1: 1

Enter first CPU burst for Job 1: 3

Enter arrival time for Job 2: 2

Enter first CPU burst for Job 2: 1

Enter arrival time for Job 3: 3

Enter first CPU burst for Job 3: 1

Job Turnaround Time Waiting Time

Job 1 5 2

Job 2 7 6

Job 3 9 8

Average Turnaround Time: 7.00

Average Waiting Time: 5.33

**Set B:**

**1. Write the program to simulate Preemptive Priority scheduling. The arrival time and**

**firstCPU-burst and priority for different n number of processes should be input to**

**thealgorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should**

**begenerated randomly. The output should give Gantt chart, turnaround time and**

**waitingtime for each process. Also find the average waiting time and turnaround time.**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_PROCESSES 30

#define IO\_WAIT\_TIME 2

typedef struct {

int id;

int arrival\_time;

int burst\_time;

int priority;

int remaining\_burst\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

} Process;

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

srand(time(0)); // Seed the random number generator

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

// Generate a random burst time between 1 and 10 units

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

}

}

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

int time = 0;

int completed = 0;

int min\_priority;

int current\_process = -1;

int io\_wait = IO\_WAIT\_TIME;

while (completed < n) {

min\_priority = 1000000; // Initialize to a large value

int process\_found = 0;

// Find the process with the highest priority (lowest priority number) that has arrived

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

if (processes[i].arrival\_time <= time && processes[i].remaining\_burst\_time > 0) {

if (processes[i].priority < min\_priority) {

min\_priority = processes[i].priority;

current\_process = i;

process\_found = 1;

}

}

}

if (process\_found) {

// Execute the current process

processes[current\_process].remaining\_burst\_time--;

time++;

// If the process is completed

if (processes[current\_process].remaining\_burst\_time == 0) {

processes[current\_process].completion\_time = time;

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

processes[current\_process].waiting\_time = processes[current\_process].turnaround\_time - processes[current\_process].burst\_time;

completed++;

}

} else {

time++; // If no process is ready, just increment time

}

// Simulate I/O wait time (not directly affecting process state)

if (time % io\_wait == 0) {

// This is just a simulation of I/O wait; it doesn't affect the process directly

}

}

}

void calculateAverageTimes(Process processes[], int n, float \*avg\_waiting\_time, float \*avg\_turnaround\_time) {

int total\_waiting\_time = 0;

int total\_turnaround\_time = 0;

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

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

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

}

\*avg\_waiting\_time = (float)total\_waiting\_time / n;

\*avg\_turnaround\_time = (float)total\_turnaround\_time / n;

}

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

printf("Gantt Chart:\n");

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

printf("P[%d] ", processes[i].id);

if (i < n - 1) {

printf("-> ");

}

}

printf("\n");

}

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

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

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

printf("P[%d]\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].priority,

processes[i].waiting\_time, processes[i].turnaround\_time);

}

}

int main() {

int n;

Process processes[MAX\_PROCESSES];

float avg\_waiting\_time, avg\_turnaround\_time;

printf("Enter the number of processes (maximum %d): ", MAX\_PROCESSES);

scanf("%d", &n);

if (n <= 0 || n > MAX\_PROCESSES) {

printf("Invalid number of processes. Please enter a number between 1 and %d.\n", MAX\_PROCESSES);

return 1;

}

// Input process details

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

processes[i].id = i + 1;

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

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

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

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

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

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

}

// Generate random burst times for future bursts (not needed for this simulation)

generateRandomBurstTimes(processes, n);

// Perform the Preemptive Priority Scheduling

preemptivePriorityScheduling(processes, n);

// Calculate average waiting time and turnaround time

calculateAverageTimes(processes, n, &avg\_waiting\_time, &avg\_turnaround\_time);

// Print results

printGanttChart(processes, n);

printProcessDetails(processes, n);

printf("\nAverage Waiting Time: %.2f", avg\_waiting\_time);

printf("\nAverage Turnaround Time: %.2f\n", avg\_turnaround\_time);

return 0;

}

**Output :**

Enter the number of processes (maximum 30): 5

Enter Arrival Time for Process 1: 1

Enter Burst Time for Process 1: 3

Enter Priority for Process 1: 2

Enter Arrival Time for Process 2: 3

Enter Burst Time for Process 2: 4

Enter Priority for Process 2: 1

Enter Arrival Time for Process 3: 2

Enter Burst Time for Process 3: 2

Enter Priority for Process 3: 2

Enter Arrival Time for Process 4: 4

Enter Burst Time for Process 4: 6

Enter Priority for Process 4: 1

Enter Arrival Time for Process 5: 2

Enter Burst Time for Process 5: 3

Enter Priority for Process 5: 4

Gantt Chart:

P[1] -> P[2] -> P[3] -> P[4] -> P[5]

Process Arrival Time Burst Time Priority Waiting Time Turnaround Time

P[1] 1 3 2 10 13

P[2] 3 4 1 0 4

P[3] 2 2 2 12 14

P[4] 4 6 1 3 9

P[5] 2 3 4 14 17

Average Waiting Time: 7.80

Average Turnaround Time: 11.40

**2. Write the program to simulate Non-preemptive Priority scheduling. The arrival time**

**andfirst CPU-burst and priority for different n number of processes should be input to**

**thealgorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should**

**begenerated randomly. The output should give Gantt chart, turnaround time and**

**waitingtime for each process. Also find the average waiting time and turnaround time.**

**Ans**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

typedef struct {

char name[10];

int arrival\_time;

int cpu\_burst;

int priority;

int waiting\_time;

int turnaround\_time;

} Process;

int generate\_next\_burst() {

return rand() % 10 + 1; }

void priority\_scheduling(Process processes[], int num\_processes) {

int time = 0, completed = 0;

bool finished[num\_processes];

for (int i = 0; i < num\_processes; i++) {

finished[i] = false;

}

while (completed < num\_processes) {

int highest\_priority = -1;

int index = -1;

for (int i = 0; i < num\_processes; i++) {

if (processes[i].arrival\_time <= time && !finished[i]) {

if (highest\_priority == -1 || processes[i].priority < processes[highest\_priority].priority) {

highest\_priority = i;

index = highest\_priority;

}

}

}

if (index != -1) {

time += processes[index].cpu\_burst + 2; // Add CPU burst and fixed I/O waiting time

processes[index].turnaround\_time = time - processes[index].arrival\_time;

processes[index].waiting\_time = processes[index].turnaround\_time - processes[index].cpu\_burst;

finished[index] = true;

completed++;

processes[index].cpu\_burst = generate\_next\_burst(); // Generate the next CPU burst

} else {

time++;

}

}

}

int main() {

int num\_processes;

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

scanf("%d", &num\_processes);

Process processes[num\_processes];

for (int i = 0; i < num\_processes; i++) {

printf("Enter arrival time for Process %d: ", i + 1);

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

printf("Enter first CPU burst for Process %d: ", i + 1);

scanf("%d", &processes[i].cpu\_burst);

printf("Enter priority for Process %d (lower number indicates higher priority): ", i + 1);

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

snprintf(processes[i].name, sizeof(processes[i].name), "P%d", i + 1);

}

priority\_scheduling(processes, num\_processes);

printf("\nJob\tTurnaround Time\tWaiting Time\n");

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

for (int i = 0; i < num\_processes; i++) {

printf("%s\t%d\t\t%d\n", processes[i].name, processes[i].turnaround\_time, processes[i].waiting\_time);

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

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

}

printf("\nAverage Turnaround Time: %.2f\n", (float)total\_turnaround\_time / num\_processes);

printf("Average Waiting Time: %.2f\n", (float)total\_waiting\_time / num\_processes);

return 0;

}

**Output**

Enter the number of processes: 5

Enter arrival time for Process 1: 2

Enter first CPU burst for Process 1: 3

Enter priority for Process 1 (lower number indicates higher priority): 2

Enter arrival time for Process 2: 2

Enter first CPU burst for Process 2: 5

Enter priority for Process 2 (lower number indicates higher priority): 1

Enter arrival time for Process 3: 2

Enter first CPU burst for Process 3: 1

Enter priority for Process 3 (lower number indicates higher priority): 4

Enter arrival time for Process 4: 2

Enter first CPU burst for Process 4: 4

Enter priority for Process 4 (lower number indicates higher priority): 3

Enter arrival time for Process 5: 4

Enter first CPU burst for Process 5: 2

Enter priority for Process 5 (lower number indicates higher priority): 2

Job Turnaround Time Waiting Time

P1 12 9

P2 7 2

P3 25 24

P4 22 18

P5 14 12