**­**

|  |  |
| --- | --- |
| **Total Marks:** |  |
| **Obtained Marks:** |  |

**Operating System Lab**

**Class Task**

**Last date of Submission: April 27, 2025**

**Submitted To: Sir Jawad Naseer**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Student Name: Ubaid Bin Waris**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Reg Number: 2212416**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**First Come First Serve**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define NUM\_PROCESSES 5

typedef struct {

int pid;

int arrival\_time;

int burst\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

} Process;

void swap(Process \*a, Process \*b) {

Process temp = \*a;

\*a = \*b;

\*b = temp;

}

// Sort by arrival time

void sort\_by\_arrival(Process p[], int n) {

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

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

if (p[j].arrival\_time > p[j+1].arrival\_time) {

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

}

}

}

}

int main() {

Process p[NUM\_PROCESSES];

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

float throughput;

// Random seed

srand(0);

// Randomly generate arrival and burst times

printf("Randomly Generated Processes:\n");

printf("PID\tArrival Time\tBurst Time\n");

for (int i = 0; i < NUM\_PROCESSES; i++) {

p[i].pid = i + 1;

p[i].arrival\_time = rand() % 10; // Random arrival time between 0 and 9

p[i].burst\_time = (rand() % 5) + 1; // Random burst time between 1 and 5

printf("P%d\t%d\t\t%d\n", p[i].pid, p[i].arrival\_time, p[i].burst\_time);

}

// Sort processes according to arrival time

sort\_by\_arrival(p, NUM\_PROCESSES);

// Calculate completion, turnaround, and waiting times

int current\_time = 0;

for (int i = 0; i < NUM\_PROCESSES; i++) {

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

current\_time = p[i].arrival\_time; // CPU remains idle

}

p[i].completion\_time = current\_time + p[i].burst\_time;

p[i].turnaround\_time = p[i].completion\_time - p[i].arrival\_time;

p[i].waiting\_time = p[i].turnaround\_time - p[i].burst\_time;

current\_time = p[i].completion\_time;

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

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

}

throughput = (float)NUM\_PROCESSES / (float)current\_time;

printf("\n\n--- FCFS Scheduling Results ---\n");

printf("PID\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");

for (int i = 0; i < NUM\_PROCESSES; i++) {

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

p[i].pid,

p[i].arrival\_time,

p[i].burst\_time,

p[i].completion\_time,

p[i].turnaround\_time,

p[i].waiting\_time

);

}

printf("\nExecution Order: ");

for (int i = 0; i < NUM\_PROCESSES; i++) {

printf("P%d", p[i].pid);

if (i != NUM\_PROCESSES-1) {

printf(" -> ");

}

}

printf("\n\nTotal Time: %d", current\_time);

printf("\nThroughput: %.2f processes/unit time", throughput);

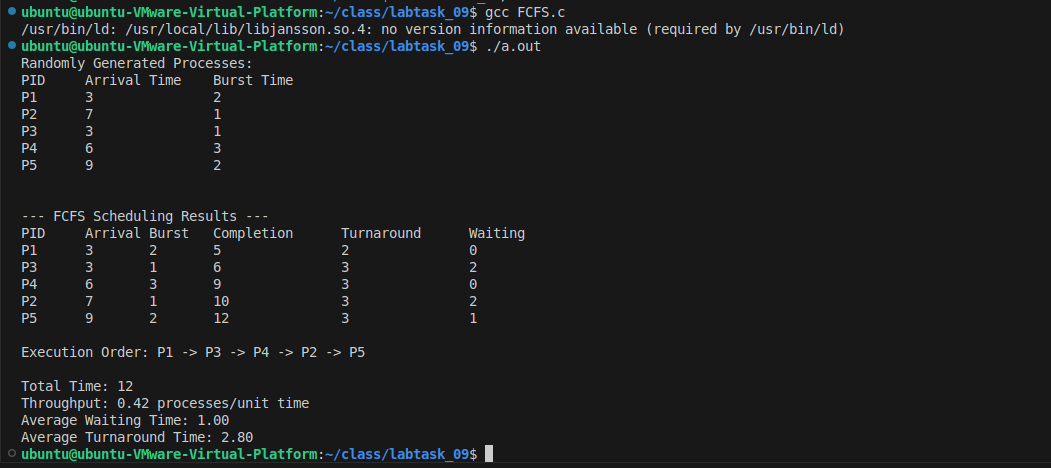
printf("\nAverage Waiting Time: %.2f", (float)total\_waiting\_time / NUM\_PROCESSES);

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

return 0;

}

**OUTPUT:**



**Shortest Job Next**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define NUM\_PROCESSES 5

typedef struct {

int pid;

int arrival\_time;

int burst\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int is\_completed;

} Process;

void swap(Process \*a, Process \*b) {

Process temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

Process p[NUM\_PROCESSES];

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

float throughput;

srand(0); // fixed seed for reproducibility

// Randomly generate arrival and burst times

printf("Randomly Generated Processes:\n");

printf("PID\tArrival Time\tBurst Time\n");

for (int i = 0; i < NUM\_PROCESSES; i++) {

p[i].pid = i + 1;

p[i].arrival\_time = rand() % 10; // Random arrival time between 0 and 9

p[i].burst\_time = (rand() % 5) + 1; // Random burst time between 1 and 5

p[i].is\_completed = 0; // Initially, no process is completed

printf("P%d\t%d\t\t%d\n", p[i].pid, p[i].arrival\_time, p[i].burst\_time);

}

int completed = 0, current\_time = 0;

printf("\nExecution Order: ");

while (completed != NUM\_PROCESSES) {

int idx = -1;

int min\_burst = 1e9; // Large number

// Find process with shortest burst among arrived processes

for (int i = 0; i < NUM\_PROCESSES; i++) {

if (p[i].arrival\_time <= current\_time && p[i].is\_completed == 0) {

if (p[i].burst\_time < min\_burst) {

min\_burst = p[i].burst\_time;

idx = i;

}

else if (p[i].burst\_time == min\_burst) {

// Tie breaker: earlier arrival time

if (p[i].arrival\_time < p[idx].arrival\_time) {

idx = i;

}

}

}

}

if (idx != -1) {

// Execute the selected process

p[idx].completion\_time = current\_time + p[idx].burst\_time;

p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

p[idx].is\_completed = 1;

total\_waiting\_time += p[idx].waiting\_time;

total\_turnaround\_time += p[idx].turnaround\_time;

current\_time = p[idx].completion\_time;

completed++;

printf("P%d", p[idx].pid);

if (completed != NUM\_PROCESSES) {

printf(" -> ");

}

}

else {

// No process has arrived yet, move time forward

current\_time++;

}

}

throughput = (float)NUM\_PROCESSES / (float)current\_time;

printf("\n\n--- Shortest Job Next (SJN/SJF) Scheduling Results ---\n");

printf("PID\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");

for (int i = 0; i < NUM\_PROCESSES; i++) {

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

p[i].pid,

p[i].arrival\_time,

p[i].burst\_time,

p[i].completion\_time,

p[i].turnaround\_time,

p[i].waiting\_time

);

}

printf("\nTotal Time: %d", current\_time);

printf("\nThroughput: %.2f processes/unit time", throughput);

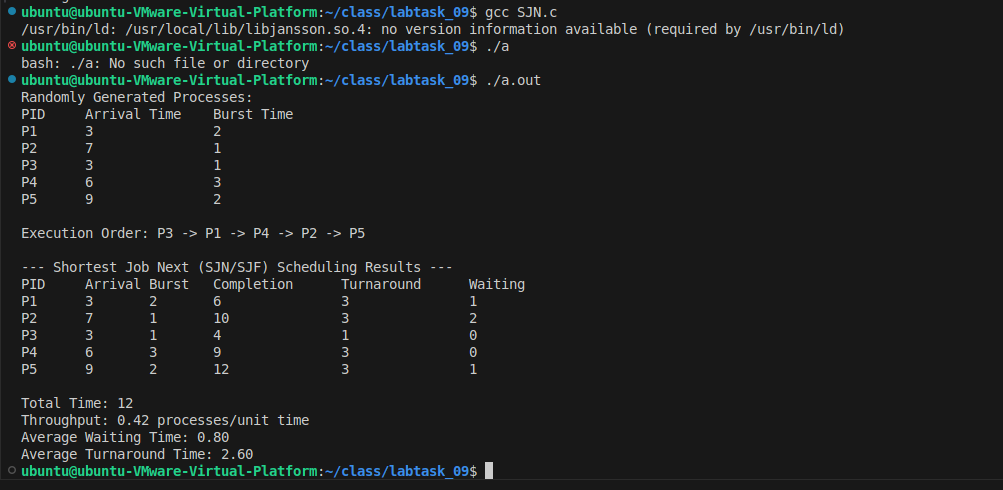
printf("\nAverage Waiting Time: %.2f", (float)total\_waiting\_time / NUM\_PROCESSES);

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

return 0;

}

**OUTPUT:**



**Priority algorithm**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define NUM\_PROCESSES 5

typedef struct {

int pid;

int arrival\_time;

int burst\_time;

int priority;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int is\_completed;

} Process;

void swap(Process \*a, Process \*b) {

Process temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

Process p[NUM\_PROCESSES];

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

float throughput;

srand(1); // fixed seed for reproducibility

// Randomly generate arrival time, burst time, and priority

printf("Randomly Generated Processes:\n");

printf("PID\tArrival Time\tBurst Time\tPriority\n");

for (int i = 0; i < NUM\_PROCESSES; i++) {

p[i].pid = i + 1;

p[i].arrival\_time = rand() % 10; // Arrival time between 0 and 9

p[i].burst\_time = (rand() % 5) + 1; // Burst time between 1 and 5

p[i].priority = (rand() % 5) + 1; // Priority between 1 and 5 (1 highest)

p[i].is\_completed = 0; // Initially not completed

printf("P%d\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].arrival\_time, p[i].burst\_time, p[i].priority);

}

int completed = 0, current\_time = 0;

printf("\nExecution Order: ");

while (completed != NUM\_PROCESSES) {

int idx = -1;

int highest\_priority = 1e9; // Smaller number means higher priority

// Find process with highest priority among arrived processes

for (int i = 0; i < NUM\_PROCESSES; i++) {

if (p[i].arrival\_time <= current\_time && p[i].is\_completed == 0) {

if (p[i].priority < highest\_priority) {

highest\_priority = p[i].priority;

idx = i;

}

else if (p[i].priority == highest\_priority) {

// Tie breaker: earlier arrival time

if (p[i].arrival\_time < p[idx].arrival\_time) {

idx = i;

}

}

}

}

if (idx != -1) {

// Execute the selected process

p[idx].completion\_time = current\_time + p[idx].burst\_time;

p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

p[idx].is\_completed = 1;

total\_waiting\_time += p[idx].waiting\_time;

total\_turnaround\_time += p[idx].turnaround\_time;

current\_time = p[idx].completion\_time;

completed++;

printf("P%d", p[idx].pid);

if (completed != NUM\_PROCESSES) {

printf(" -> ");

}

}

else {

// No process has arrived yet, move time forward

current\_time++;

}

}

throughput = (float)NUM\_PROCESSES / (float)current\_time;

printf("\n\n--- Priority Scheduling (Non-Preemptive) Results ---\n");

printf("PID\tArrival\tBurst\tPriority\tCompletion\tTurnaround\tWaiting\n");

for (int i = 0; i < NUM\_PROCESSES; i++) {

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

p[i].pid,

p[i].arrival\_time,

p[i].burst\_time,

p[i].priority,

p[i].completion\_time,

p[i].turnaround\_time,

p[i].waiting\_time

);

}

printf("\nTotal Time: %d", current\_time);

printf("\nThroughput: %.2f processes/unit time", throughput);

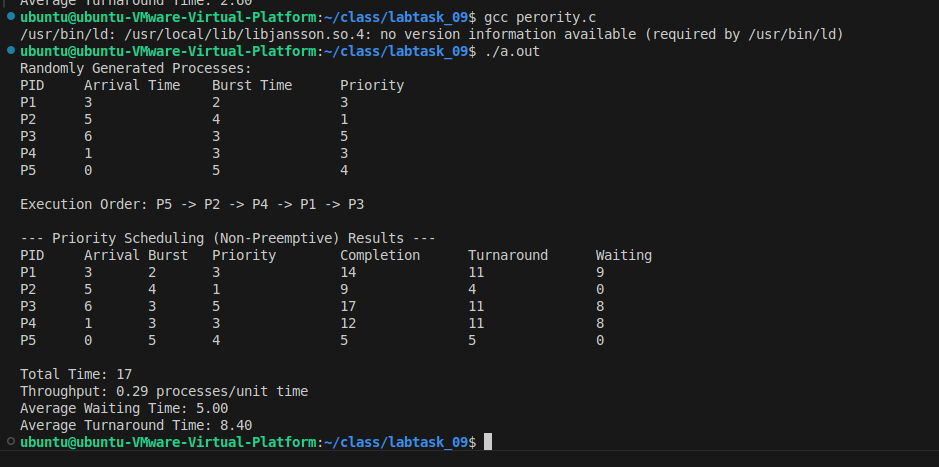
printf("\nAverage Waiting Time: %.2f", (float)total\_waiting\_time / NUM\_PROCESSES);

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

return 0;

}

**OUTPUT:**



**Round Robin**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define NUM\_PROCESSES 5

#define TIME\_QUANTUM 3

typedef struct {

int pid;

int arrival\_time;

int burst\_time;

int remaining\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

} Process;

int main() {

Process p[NUM\_PROCESSES];

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

float throughput;

srand(2); // fixed seed for reproducibility

// Randomly generate arrival time and burst time

printf("Randomly Generated Processes:\n");

printf("PID\tArrival Time\tBurst Time\n");

for (int i = 0; i < NUM\_PROCESSES; i++) {

p[i].pid = i + 1;

p[i].arrival\_time = rand() % 10; // Arrival time between 0 and 9

p[i].burst\_time = (rand() % 7) + 1; // Burst time between 1 and 7

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

printf("P%d\t%d\t\t%d\n", p[i].pid, p[i].arrival\_time, p[i].burst\_time);

}

int completed = 0, current\_time = 0;

int queue[NUM\_PROCESSES];

int front = 0, rear = 0;

int visited[NUM\_PROCESSES] = {0};

printf("\nExecution Order: ");

while (completed != NUM\_PROCESSES) {

// Enqueue newly arrived processes

for (int i = 0; i < NUM\_PROCESSES; i++) {

if (p[i].arrival\_time <= current\_time && visited[i] == 0) {

queue[rear++] = i;

visited[i] = 1;

}

}

if (front == rear) {

// CPU idle

current\_time++;

continue;

}

int idx = queue[front++];

if (p[idx].remaining\_time <= TIME\_QUANTUM) {

current\_time += p[idx].remaining\_time;

p[idx].remaining\_time = 0;

p[idx].completion\_time = current\_time;

completed++;

printf("P%d ", p[idx].pid);

// Enqueue new processes during execution

for (int i = 0; i < NUM\_PROCESSES; i++) {

if (p[i].arrival\_time <= current\_time && visited[i] == 0) {

queue[rear++] = i;

visited[i] = 1;

}

}

}

else {

current\_time += TIME\_QUANTUM;

p[idx].remaining\_time -= TIME\_QUANTUM;

printf("P%d ", p[idx].pid);

// Enqueue new processes during execution

for (int i = 0; i < NUM\_PROCESSES; i++) {

if (p[i].arrival\_time <= current\_time && visited[i] == 0) {

queue[rear++] = i;

visited[i] = 1;

}

}

// Put current process back in queue

queue[rear++] = idx;

}

}

printf("\n\n--- Round Robin Scheduling (Time Quantum = %d) Results ---\n", TIME\_QUANTUM);

printf("PID\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");

for (int i = 0; i < NUM\_PROCESSES; i++) {

p[i].turnaround\_time = p[i].completion\_time - p[i].arrival\_time;

p[i].waiting\_time = p[i].turnaround\_time - p[i].burst\_time;

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

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

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

p[i].pid,

p[i].arrival\_time,

p[i].burst\_time,

p[i].completion\_time,

p[i].turnaround\_time,

p[i].waiting\_time

);

}

throughput = (float)NUM\_PROCESSES / (float)current\_time;

printf("\nTotal Time: %d", current\_time);

printf("\nThroughput: %.2f processes/unit time", throughput);

printf("\nAverage Waiting Time: %.2f", (float)total\_waiting\_time / NUM\_PROCESSES);

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

return 0;

**OUTPUT:**

