**Week 1**

**Name: Adarsh Tomar**

**Roll no-02**

**Section-A1**

**Q-1** Write a program to create a child process using system call fork().

**Source Code-:**

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

printf("Fork failed!\n");

return 1;

}

else if (pid == 0) {

printf("Hello from the Child Process! PID: %d\n", getpid());

}

else {

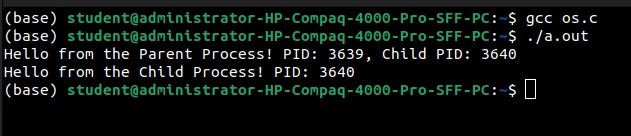
printf("Hello from the Parent Process! PID: %d, Child PID: %d\n", getpid(), pid);

}

return 0;

}

**OUTPUT :**



**Q-2** Write a program to print process Id's of parent and child process i.e. parent should print its own and its child process id while child process should print its own and its parent process id. (use getpid(), getppid())

**Source Code-:**

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

printf("Fork failed!\n");

return 1;

}

else if (pid == 0) {

printf("Child Process:\n");

printf("PID: %d, Parent PID: %d\n", getpid(), getppid());

}

else {

printf("Parent Process:\n");

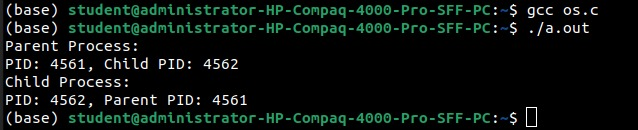
printf("PID: %d, Child PID: %d\n", getpid(), pid);

}

return 0;

}

**OUTPUT :**



**Q-3** Write a program to create child process which will list all the files present in your system. Make sure that parent process waits until child has not completed its execution. (use wait(), exit()) What will happen if parent process dies before child process? Illustrate it by creating one more child of parent process.

**Source Code-:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

pid\_t pid1, pid2;

pid1 = fork();

if (pid1 < 0) {

printf("Fork failed!\n");

return 1;

} else if (pid1 == 0) {

printf("Child Process 1 (PID: %d): Listing files...\n", getpid());

execlp("ls", "ls", "-l", (char \*)NULL);

exit(0);

}

else {

wait(NULL);

printf("Parent Process (PID: %d): First child completed.\n", getpid());

pid2 = fork();

if (pid2 < 0) {

printf("Fork failed!\n");

return 1;

}

else if (pid2 == 0) {

printf("Child Process 2 (PID: %d): I am the second child.\n", getpid());

sleep(5);

printf("Child Process 2 (PID: %d): Work done.\n", getpid());

exit(0);

}

else {

printf("Parent Process (PID: %d): Exiting now.\n", getpid());

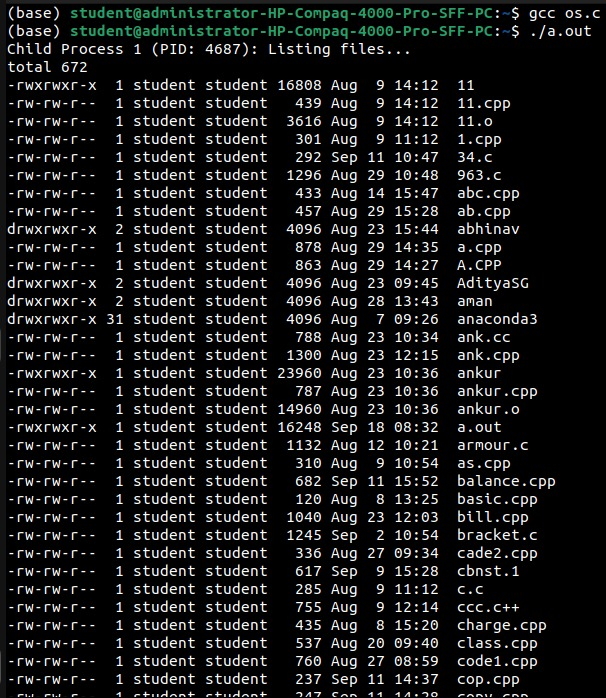
exit(0);

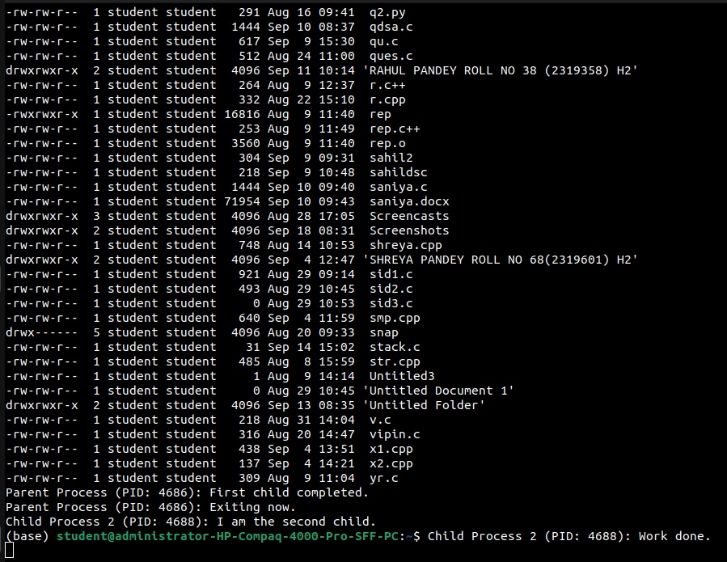
}

}

}

**OUTPUT :**





**Week 2**

**Name: Adarsh Tomar**

**Roll no-02**

**Section-A1**

**Q-1** Write a program to open a directory and list its contents. (use opendir(), readdir(), closedir() )

**Source Code-:**

#include <stdio.h>

#include <dirent.h>

int main() {

struct dirent \*de;

DIR \*dr = opendir(".");

if (dr == NULL) {

printf("Could not open directory.\n");

return 1;

}

while ((de = readdir(dr)) != NULL) {

printf("%s\t", de->d\_name);

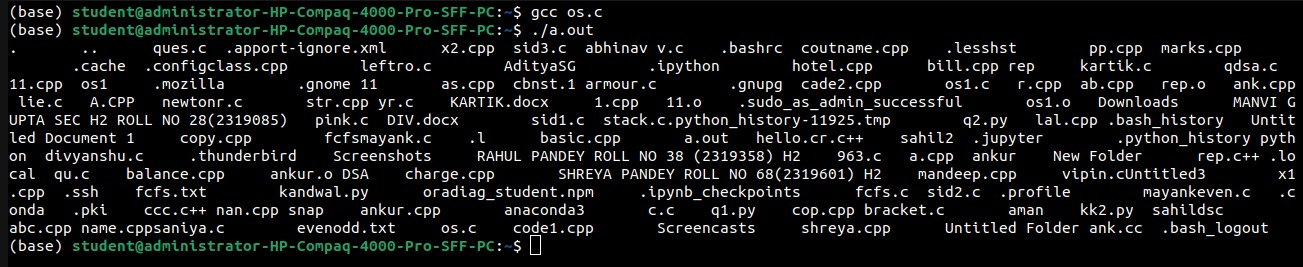
}

closedir(dr);

return 0;

}

**OUTPUT :**



**Q-2** . Write a program to show working of execlp() system call by executing ls command.

**Source Code-:**

#include <stdio.h>

#include <unistd.h>

int main() {

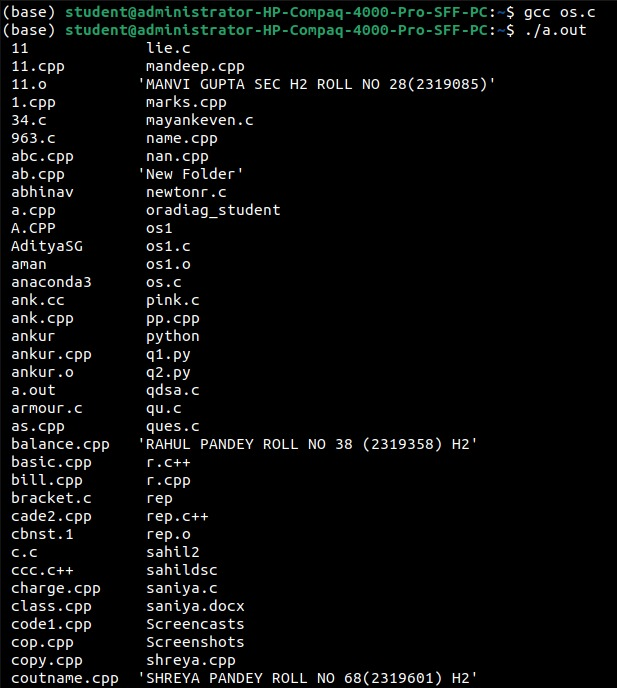
execlp("ls", "ls", NULL);

perror("execlp failed");

return 1;

}

**OUTPUT :**



**Q-3** Write a program to read a file and store your details in that file. Your program should also create one more file and store your friends details in that file. Once both files are created, print lines which are matching in both files.

**Source Code-:**

#include <stdio.h>

#include <string.h>

int main() {

FILE \*file1, \*file2;

char line1[100], line2[100];

file1 = fopen("my\_details.txt", "w");

fprintf(file1, "Alice\nBob\nCharlie\n");

fclose(file1);

file2 = fopen("friends\_details.txt", "w");

fprintf(file2, "Charlie\nDavid\nEve\n");

fclose(file2);

file1 = fopen("my\_details.txt", "r");

file2 = fopen("friends\_details.txt", "r");

if (file1 == NULL || file2 == NULL) {

printf("Error opening files.\n");

return 1;

}

while (fgets(line1, sizeof(line1), file1)) {

fseek(file2, 0, SEEK\_SET);

while (fgets(line2, sizeof(line2), file2)) {

if (strcmp(line1, line2) == 0) {

printf("Matching line: %s", line1);

break;

}

}

}

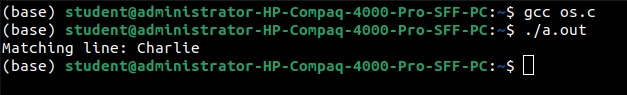
fclose(file1);

fclose(file2);

return 0;

}

**OUTPUT :**



**Week 3**

**Name: Adarsh Tomar**

**Roll no-02**

**Section-A1**

**Q-1** FCFS – First Come First Served : process which arrives first will get the CPU first.

**Source Code-:**

#include <stdio.h>

typedef struct {

int id;

int arrivalTime;

int burstTime;

int completionTime;

int turnaroundTime;

int waitingTime;

} Process;

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

int totalWaitingTime = 0, totalTurnaroundTime = 0;

processes[0].completionTime = processes[0].arrivalTime + processes[0].burstTime;

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

if (processes[i].arrivalTime > processes[i - 1].completionTime) {

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

} else {

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

}

}

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

processes[i].turnaroundTime = processes[i].completionTime - processes[i].arrivalTime;

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

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

}

printf("Gantt Chart:\n|");

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

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

}

printf("\n");

printf("0");

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

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

}

printf("\n");

printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

Process processes[n];

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

processes[i].id = i;

printf("Enter Burst Time and Arrival Time for Process P%d: ", i);

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

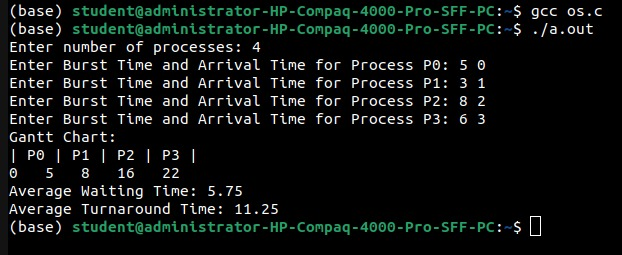
}

calculateFCFS(processes, n);

return 0;

}

**OUTPUT :**



**Q-2** SJF NP – Shortest Job First Non-Preemptive : process which needs CPU for least amount will get the CPU first. Here non-preemptive means currently running process leaves CPU voultarily after completing its execution .

**Source Code-:**

#include <stdio.h>

typedef struct {

int id;

int arrivalTime;

int burstTime;

int completionTime;

int turnaroundTime;

int waitingTime;

} Process;

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

int totalWaitingTime = 0, totalTurnaroundTime = 0;

int completed = 0, time = 0;

int minIndex, minBurstTime;

while (completed < n) {

minIndex = -1;

minBurstTime = 9999;

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

if (processes[i].arrivalTime <= time && processes[i].completionTime == 0) {

if (processes[i].burstTime < minBurstTime) {

minBurstTime = processes[i].burstTime;

minIndex = i;

}

}

}

if (minIndex == -1) {

time++;

continue;

}

processes[minIndex].completionTime = time + processes[minIndex].burstTime;

time += processes[minIndex].burstTime;

completed++;

}

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

processes[i].turnaroundTime = processes[i].completionTime - processes[i].arrivalTime;

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

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

}

printf("Gantt Chart:\n|");

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

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

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

}

}

printf("\n");

printf("0");

time = 0;

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

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

time = processes[i].completionTime;

printf(" %d", time);

}

}

printf("\n");

printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

Process processes[n];

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

processes[i].id = i;

printf("Enter Burst Time and Arrival Time for Process P%d: ", i);

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

processes[i].completionTime = 0;

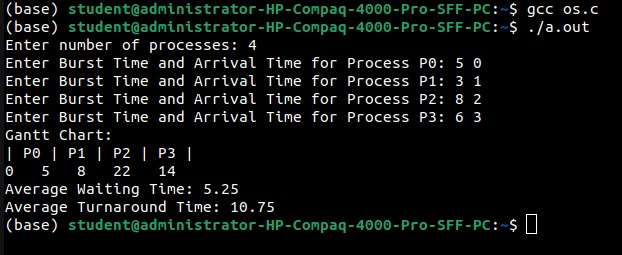
}

calculateSJFNP(processes, n);

return 0;

}

**OUTPUT :**



**Q-3** SJF P – Shortest Job First Preemptive – Here preemptive means operating system decides when to move currently running process.

**Source Code-:**

#include <stdio.h>

#include <stdbool.h>

typedef struct {

int id;

int arrivalTime;

int burstTime;

int remainingTime;

int completionTime;

int turnaroundTime;

int waitingTime;

} Process;

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

int totalWaitingTime = 0, totalTurnaroundTime = 0;

int time = 0, completed = 0;

bool isCompleted[n];

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

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

isCompleted[i] = false;

}

while (completed < n) {

int idx = -1;

int minBurstTime = 9999;

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

if (processes[i].arrivalTime <= time && !isCompleted[i] && processes[i].remainingTime < minBurstTime) {

minBurstTime = processes[i].remainingTime;

idx = i;

}

}

if (idx == -1) {

time++;

continue;

}

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

processes[idx].remainingTime--;

time++;

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

processes[idx].completionTime = time;

processes[idx].turnaroundTime = processes[idx].completionTime - processes[idx].arrivalTime;

processes[idx].waitingTime = processes[idx].turnaroundTime - processes[idx].burstTime;

totalWaitingTime += processes[idx].waitingTime;

totalTurnaroundTime += processes[idx].turnaroundTime;

isCompleted[idx] = true;

completed++;

}

}

printf("\n0");

time = 0;

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

time = processes[i].completionTime;

if (time > 0) {

printf(" %d", time);

}

}

printf("\n");

printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

Process processes[n];

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

processes[i].id = i;

printf("Enter Burst Time and Arrival Time for Process P%d: ", i);

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

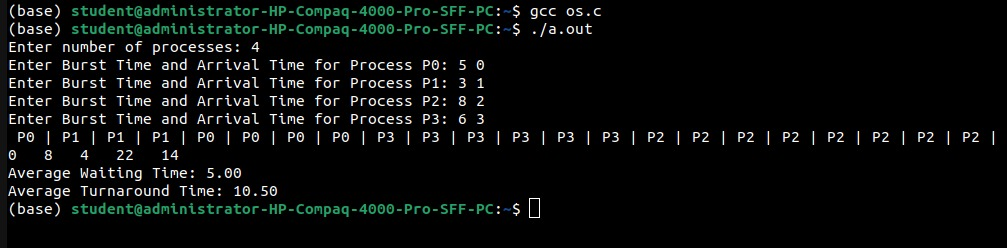
}

calculateSJFPreemptive(processes, n);

return 0;

}

**OUTPUT :**



**Week 4**

**Name: Adarsh Tomar**

**Roll no-02**

**Section-A1**

**Q-1** Priority – process which has highest priority will get CPU first.

**Source Code-:**

#include <stdio.h>

typedef struct {

int id;

int burstTime;

int arrivalTime;

int waitingTime;

int turnaroundTime;

int completionTime;

int priority;

} Process;

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

int totalWaitingTime = 0, totalTurnaroundTime = 0;

int completed = 0, time = 0;

while (completed < n) {

int idx = -1, maxPriority = -1;

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

if (processes[i].arrivalTime <= time && processes[i].completionTime == 0) {

if (processes[i].priority > maxPriority) {

maxPriority = processes[i].priority;

idx = i;

}

}

}

if (idx == -1) {

time++;

continue;

}

time += processes[idx].burstTime;

processes[idx].completionTime = time;

processes[idx].turnaroundTime = processes[idx].completionTime - processes[idx].arrivalTime;

processes[idx].waitingTime = processes[idx].turnaroundTime - processes[idx].burstTime;

totalWaitingTime += processes[idx].waitingTime;

totalTurnaroundTime += processes[idx].turnaroundTime;

completed++;

}

printf("Gantt Chart (Priority):\n|");

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

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

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

}

}

printf("\n");

printf("Average Waiting Time (Priority): %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time (Priority): %.2f\n", (float)totalTurnaroundTime / n);

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

Process processes[n];

printf("Enter Burst Times: ");

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

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

processes[i].id = i;

processes[i].completionTime = 0;

}

printf("Enter Arrival Times: ");

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

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

}

printf("Enter Priorities: ");

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

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

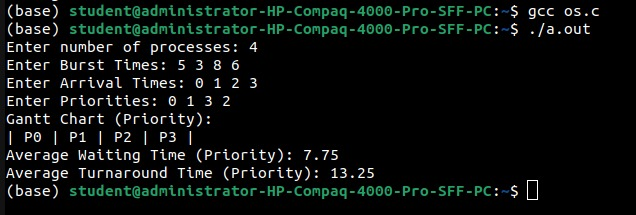
}

calculatePriority(processes, n);

return 0;

}

**OUTPUT :**



**Q-2** Round Robin – each process is provided a fix time to execute. Once a process is executed for a given time period, it is preempted and other process executes for the given time period.

**Source Code-:**

#include <stdio.h>

typedef struct {

int id;

int burstTime;

int arrivalTime;

int waitingTime;

int turnaroundTime;

int completionTime;

} Process;

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

int totalWaitingTime = 0, totalTurnaroundTime = 0;

int remainingTime[n];

for (int i = 0; i < n; i++) remainingTime[i] = processes[i].burstTime;

int completed = 0, time = 0;

printf("Gantt Chart (Round Robin):\n|");

while (completed < n) {

int done = 0;

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

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

done = 1;

if (remainingTime[i] > quantum) {

time += quantum;

remainingTime[i] -= quantum;

} else {

time += remainingTime[i];

processes[i].completionTime = time;

processes[i].turnaroundTime = processes[i].completionTime - processes[i].arrivalTime;

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

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

remainingTime[i] = 0;

completed++;

}

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

}

}

if (!done) time++;

}

printf("\n");

printf("Average Waiting Time (Round Robin): %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time (Round Robin): %.2f\n", (float)totalTurnaroundTime / n);

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

Process processes[n];

printf("Enter Burst Times: ");

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

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

processes[i].id = i;

processes[i].completionTime = 0;

}

printf("Enter Arrival Times: ");

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

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

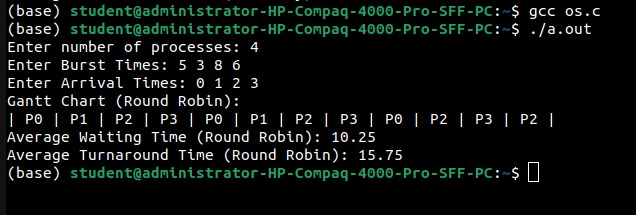
}

calculateRoundRobin(processes, n, 2);

return 0;

}

**OUTPUT :**



**Q-3** Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario: all the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

**Source Code-:**

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int id;

int burstTime;

int arrivalTime;

int waitingTime;

int turnaroundTime;

int completionTime;

} Process;

void calculateFCFS(Process processes[], int n, int \*totalWaitingTime, int \*totalTurnaroundTime) {

int time = 0;

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

if (time < processes[i].arrivalTime) {

time = processes[i].arrivalTime;

}

time += processes[i].burstTime;

processes[i].completionTime = time;

processes[i].turnaroundTime = processes[i].completionTime - processes[i].arrivalTime;

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

\*totalWaitingTime += processes[i].waitingTime;

\*totalTurnaroundTime += processes[i].turnaroundTime;

}

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

Process processes[n];

int totalWaitingTime = 0, totalTurnaroundTime = 0;

printf("Enter Burst Times: ");

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

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

processes[i].id = i;

processes[i].completionTime = 0;

}

printf("Enter Arrival Times: ");

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

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

}

int systemProcessCount;

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

scanf("%d", &systemProcessCount);

int systemProcesses[systemProcessCount];

printf("Enter IDs of system processes: ");

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

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

}

Process systemQueue[n], userQueue[n];

int sysCount = 0, userCount = 0;

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

int isSystemProcess = 0;

for (int j = 0; j < systemProcessCount; j++) {

if (processes[i].id == systemProcesses[j]) {

isSystemProcess = 1;

break;

}

}

if (isSystemProcess) {

systemQueue[sysCount++] = processes[i];

} else {

userQueue[userCount++] = processes[i];

}

}

printf("Gantt Chart:\n|");

calculateFCFS(systemQueue, sysCount, &totalWaitingTime, &totalTurnaroundTime);

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

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

}

calculateFCFS(userQueue, userCount, &totalWaitingTime, &totalTurnaroundTime);

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

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

}

printf("\n");

float averageWaitingTime = (float)totalWaitingTime / n;

float averageTurnaroundTime = (float)totalTurnaroundTime / n;

printf("Average Waiting Time: %.2f\n", averageWaitingTime);

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

return 0;

}

**OUTPUT :**

