**WEEK 1**

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

**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:**

**Question 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())**

**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 is running.\n");

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

}

else {

printf("Parent Process is running.\n");

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

}

return 0;

}

**OUTPUT:**

**Question 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**

**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);

}

}

return 0;

}

**OUTPUT:**

**WEEK 2**

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

**CODE:**

#include <stdio.h>

#include <dirent.h>

int main()

{

DIR \*dir;

struct dirent \*dirEntry;

dir = opendir(".");

if (dir)

{

printf("Contents of directory:\n");

while ((dirEntry = readdir(dir)) != NULL)

{

printf("%s\n", dirEntry->d\_name);

}

closedir(dir);

}

else

{

perror("opendir");

}

return 0;

}

**OUTPUT:**

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

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

pid\_t pid;

pid = fork();

if (pid < 0)

{

perror("Fork failed");

exit(1);

}

else if (pid == 0)

{

printf("In child process (PID: %d), executing 'ls' command...\n", getpid());

if (execlp("ls", "ls", NULL) == -1)

{

perror("execlp failed");

exit(1);

}

}

else

{

printf("In parent process (PID: %d), waiting for child to finish...\n", getpid());

wait(NULL);

printf("Child process finished, parent process continuing.\n");

}

return 0; }

**OUTPUT:**

**Question 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.**

**CODE:**

#include <stdio.h>

#include <string.h>

#define MAX 100

void compareFiles(FILE \*file1, FILE \*file2)

{

char line1[MAX], line2[MAX];

rewind(file1);

rewind(file2);

printf("\nMatching lines in both files:\n");

while (fgets(line1, MAX, file1) != NULL)

{

rewind(file2); // Rewind file2 for every line of file1

while (fgets(line2, MAX, file2) != NULL)

{

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

{

printf("%s", line1);

}

} } }

int main()

{

FILE \*myFile, \*friendFile;

char input[MAX];

myFile = fopen("myDetails.txt", "w");

if (myFile == NULL)

{

printf("Error opening file myDetails.txt\n");

return 1;

}

printf("Enter your details (type '-1' to stop):\n");

while (1)

{

fgets(input, MAX, stdin);

if (strcmp(input, "-1\n") == 0)

break;

fprintf(myFile, "%s", input);

}

fclose(myFile);

friendFile = fopen("friendDetails.txt", "w");

if (friendFile == NULL)

{

printf("Error opening file friendDetails.txt\n");

return 1;

}

printf("Enter your friend's details (type '-1' to stop):\n");

while (1)

{

fgets(input, MAX, stdin);

if (strcmp(input, "-1\n") == 0)

break;

fprintf(friendFile, "%s", input);

}

fclose(friendFile);

myFile = fopen("myDetails.txt", "r");

friendFile = fopen("friendDetails.txt", "r");

if (myFile == NULL || friendFile == NULL)

{

printf("Error reopening the files for comparison.\n");

return 1;

}

compareFiles(myFile, friendFile);

fclose(myFile);

fclose(friendFile);

return 0;

}

**OUTPUT:**

**WEEK 3**

**Question 1: Write a program to implement FCFS CPU Scheduling.**

**CODE:**

#include <stdio.h>

#include <stdlib.h>

struct Process

{

int pid; // Process ID

int at; // Arrival Time

int bt; // Burst Time

int st; // Start Time

int ct; // Completion Time

int tat; // Turn Around Time

int wt; // Waiting Time

int rt; // Response Time

};

int findMax(int a, int b)

{

return (a > b) ? a : b;

}

int comparatorAT(const void \*a, const void \*b)

{

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

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

return (p1->at - p2->at);

}

int main()

{

int n;

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

scanf("%d", &n);

struct Process p[n];

int totalTAT = 0, totalWT = 0, totalRT = 0, totalIdleTime = 0;

float cpuUtilization, throughput;

printf("Enter Arrival Time of each Process: ");

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

{

p[i].pid = i + 1;

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

}

printf("Enter Burst Time of each Process: ");

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

{

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

}

qsort(p, n, sizeof(struct Process), comparatorAT);

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

{

p[i].st = (i == 0) ? p[i].at : findMax(p[i].at, p[i - 1].ct);

p[i].ct = p[i].st + p[i].bt;

p[i].tat = p[i].ct - p[i].at;

p[i].wt = p[i].tat - p[i].bt;

p[i].rt = p[i].wt;

totalTAT += p[i].tat;

totalWT += p[i].wt;

totalRT += p[i].rt;

if (i > 0)

{

totalIdleTime += p[i].st - p[i - 1].ct;

}

}

int totalTime = p[n - 1].ct - p[0].st;

cpuUtilization = ((totalTime - totalIdleTime) / (float)totalTime) \* 100;

throughput = (float)n / totalTime;

printf("\nPID\tAT\tBT\tST\tCT\tTAT\tWT\tRT\n");

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

{

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

p[i].pid, p[i].at, p[i].bt, p[i].st, p[i].ct, p[i].tat, p[i].wt, p[i].rt);

}

printf("\nGantt chart: ");

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

{

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

}

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

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

printf("\nAverage Response Time: %.2f", (float)totalRT / n);

printf("\nCPU Utilization: %.2f%%", cpuUtilization);

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

return 0;

}

**OUTPUT:**

**Question 2: Write a program to implement SJF (Non-Preemptive) CPU Scheduling.**

**CODE:**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

struct Process

{

int pid;

int at;

int bt;

int st;

int ct;

int tat;

int wt;

int rt;

};

int findMax(int a, int b)

{

return (a > b) ? a : b;

}

int findMin(int a, int b)

{

return (a < b) ? a : b;

}

int main()

{

int n;

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

scanf("%d", &n);

struct Process p[n];

printf("Enter Arrival Time of each Process: ");

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

{

p[i].pid = i + 1;

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

}

printf("Enter Burst Time of each Process: ");

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

{

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

}

int completed = 0, currTime = 0, prev = 0;

int sumtat = 0, sumwt = 0, sumrt = 0, total\_idleTime = 0, lengthCycle = 0;

bool is\_Complete[n];

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

is\_Complete[i] = false;

int gantt[n];

int ganttIndex = 0;

while (completed != n)

{

int minidx = -1;

int mini = INT\_MAX;

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

{

if (p[i].at <= currTime && !is\_Complete[i])

{

if (p[i].bt < mini)

{

mini = p[i].bt;

minidx = i;

}

if (p[i].bt == mini && p[i].at < p[minidx].at)

{

mini = p[i].bt;

minidx = i;

}

}

}

if (minidx == -1)

currTime++;

else

{

p[minidx].st = currTime;

p[minidx].ct = p[minidx].st + p[minidx].bt;

p[minidx].tat = p[minidx].ct - p[minidx].at;

p[minidx].wt = p[minidx].tat - p[minidx].bt;

p[minidx].rt = p[minidx].wt;

sumtat += p[minidx].tat;

sumwt += p[minidx].wt;

sumrt += p[minidx].rt;

total\_idleTime += (ganttIndex == 0) ? 0 : p[minidx].st - prev;

gantt[ganttIndex++] = p[minidx].pid;

completed++;

currTime = p[minidx].ct;

prev = currTime;

is\_Complete[minidx] = true;

}

}

int max\_completionTime = INT\_MIN;

int min\_ArrivalTime = INT\_MAX;

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

{

max\_completionTime = findMax(max\_completionTime, p[i].ct);

min\_ArrivalTime = findMin(min\_ArrivalTime, p[i].at);

}

lengthCycle = max\_completionTime - min\_ArrivalTime;

float cpu\_Utilization = (float)(lengthCycle - total\_idleTime) \* 100 / lengthCycle;

float throughput = (float)n / lengthCycle;

printf("Gantt's Chart: ");

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

{

printf("P%d ", gantt[i]);

}

printf("\n");

printf("Total Turn Around Time = %d\n", sumtat);

printf("Total Waiting Time = %d\n", sumwt);

printf("Total Response Time = %d\n", sumrt);

printf("Total Idle Time = %d\n", total\_idleTime);

printf("Length Cycle = %d\n", lengthCycle);

printf("Average Turn Around Time = %.2f\n", (float)sumtat / n);

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

printf("Average Response Time = %.2f\n", (float)sumrt / n);

printf("PID\tAT\tBT\tST\tCT\tTAT\tWT\tRT\n");

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

{

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].st, p[i].ct, p[i].tat, p[i].wt, p[i].rt);

}

printf("CPU Utilization = %.2f\n", cpu\_Utilization);

printf("Throughput = %.2f\n", throughput);

return 0;

}

**OUTPUT:**

**Question 3: Write a program to implement SJF (Preemptive) CPU Scheduling.**

**CODE:**

#include<stdio.h>

#include<stdbool.h>

#include<limits.h>

struct process\_struct {

int pid;

int at;

int bt;

int ct, wt, tat, rt, start\_time;

} ps[100];

int findmax(int a, int b) {

return a > b ? a : b;

}

int findmin(int a, int b) {

return a < b ? a : b;

}

int main() {

int n;

float bt\_remaining[100];

bool is\_completed[100] = {false}, is\_first\_process = true;

int current\_time = 0, completed = 0;

float sum\_tat = 0, sum\_wt = 0, sum\_rt = 0, total\_idle\_time = 0, length\_cycle, prev = 0,cpu\_utilization;

int max\_completion\_time, min\_arrival\_time,gantt\_index = 0;

int gantt\_chart[1000]; // Array to store Gantt chart

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

scanf("%d", &n);

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

printf("\nEnter Process %d Arrival Time: ", i);

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

ps[i].pid = i;

}

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

printf("\nEnter Process %d Burst Time: ", i);

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

bt\_remaining[i] = ps[i].bt;

}

while (completed != n) {

// Find process with min. burst time in the ready queue at current time

int min\_index = -1;

int minimum = INT\_MAX;

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

if (ps[i].at <= current\_time && !is\_completed[i]) {

if (bt\_remaining[i] < minimum) {

minimum = bt\_remaining[i];

min\_index = i;

}

if (bt\_remaining[i] == minimum) {

if (ps[i].at < ps[min\_index].at) {

minimum = bt\_remaining[i];

min\_index = i;

}

}

}

}

if (min\_index == -1) {

current\_time++;

} else {

if (bt\_remaining[min\_index] == ps[min\_index].bt) {

ps[min\_index].start\_time = current\_time;

total\_idle\_time += (is\_first\_process == true) ? 0 : (ps[min\_index].start\_time - prev);

is\_first\_process = false;

}

bt\_remaining[min\_index] -= 1;

gantt\_chart[gantt\_index++] = ps[min\_index].pid; // Store process ID in Gantt chart

current\_time++;

prev = current\_time;

if (bt\_remaining[min\_index] == 0) {

ps[min\_index].ct = current\_time;

ps[min\_index].tat = ps[min\_index].ct - ps[min\_index].at;

ps[min\_index].wt = ps[min\_index].tat - ps[min\_index].bt;

ps[min\_index].rt = ps[min\_index].start\_time - ps[min\_index].at;

sum\_tat += ps[min\_index].tat;

sum\_wt += ps[min\_index].wt;

sum\_rt += ps[min\_index].rt;

completed++;

is\_completed[min\_index] = true;

}

}

}

max\_completion\_time = INT\_MIN;

min\_arrival\_time = INT\_MAX;

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

max\_completion\_time = findmax(max\_completion\_time, ps[i].ct);

min\_arrival\_time = findmin(min\_arrival\_time, ps[i].at);

}

length\_cycle = max\_completion\_time - min\_arrival\_time;

printf("\nProcess No.\tAT\tCPU Burst Time\tCT\tTAT\tWT\tRT\n");

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

printf("%d\t\t%d\t%d\t\t%d\t%d\t%d\t%d\n", ps[i].pid, ps[i].at, ps[i].bt, ps[i].ct, ps[i].tat, ps[i].wt, ps[i].rt);

}

// Printing Gantt Chart

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

for (int i = 0; i < gantt\_index; i++) {

printf(" P%d |", gantt\_chart[i]);

}

printf("\n");

cpu\_utilization = (float)(length\_cycle - total\_idle\_time) / length\_cycle;

printf("\nAverage Turn Around time = %f", (float)sum\_tat / n);

printf("\nAverage Waiting Time = %f", (float)sum\_wt / n);

printf("\nAverage Response Time = %f", (float)sum\_rt / n);

printf("\nThroughput = %f", n / (float)length\_cycle);

printf("\nCPU Utilization(Percentage) = %f\n", cpu\_utilization \* 100);

}

**OUTPUT:**

**WEEK 4**

**Question 1: Write a program to implement Round Robin CPU Scheduling.**

**CODE:**

#include<stdio.h>

#include<limits.h>

#include<stdbool.h>

#include <stdlib.h>

struct process\_struct {

int pid;

int at;

int bt;

int ct, wt, tat, rt, start\_time;

int bt\_remaining;

} ps[100];

int findmax(int a, int b) {

return a > b ? a : b;

}

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

int x = ((struct process\_struct \*)a)->at;

int y = ((struct process\_struct \*)b)->at;

return (x < y) ? -1 : 1;

}

int compare\_pid(const void \*a, const void \*b) {

int x = ((struct process\_struct \*)a)->pid;

int y = ((struct process\_struct \*)b)->pid;

return (x < y) ? -1 : (x > y) ? 1 : 0;

}

int main() {

int n, index;

float cpu\_utilization;

bool visited[100] = { false }, is\_first\_process = true;

int current\_time = 0, max\_completion\_time;

int completed = 0, tq, total\_idle\_time = 0, length\_cycle;

int queue[100], front = -1, rear = -1;

float sum\_tat = 0, sum\_wt = 0, sum\_rt = 0;

int gantt\_chart[1000], gc\_index = 0; // Array to store Gantt chart

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

scanf("%d", &n);

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

printf("\nEnter Process %d Arrival Time: ", i);

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

ps[i].pid = i;

}

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

printf("\nEnter Process %d Burst Time: ", i);

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

ps[i].bt\_remaining = ps[i].bt;

}

printf("\nEnter time quantum: ");

scanf("%d", &tq);

qsort(ps, n, sizeof(struct process\_struct), comparatorAT);

front = rear = 0;

queue[rear] = 0;

visited[0] = true;

while (completed != n) {

index = queue[front++];

if (ps[index].bt\_remaining == ps[index].bt) {

ps[index].start\_time = findmax(current\_time, ps[index].at);

total\_idle\_time += (is\_first\_process) ? 0 : ps[index].start\_time - current\_time;

current\_time = ps[index].start\_time;

is\_first\_process = false;

}

gantt\_chart[gc\_index++] = ps[index].pid; // Record process execution in Gantt chart

if (ps[index].bt\_remaining - tq > 0) {

ps[index].bt\_remaining -= tq;

current\_time += tq;

} else {

current\_time += ps[index].bt\_remaining;

ps[index].bt\_remaining = 0;

completed++;

ps[index].ct = current\_time;

ps[index].tat = ps[index].ct - ps[index].at;

ps[index].wt = ps[index].tat - ps[index].bt;

ps[index].rt = ps[index].start\_time - ps[index].at;

sum\_tat += ps[index].tat;

sum\_wt += ps[index].wt;

sum\_rt += ps[index].rt;

}

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

if (ps[i].bt\_remaining > 0 && ps[i].at <= current\_time && !visited[i]) {

queue[++rear] = i;

visited[i] = true;

}

}

if (ps[index].bt\_remaining > 0) {

queue[++rear] = index;

}

if (front > rear) {

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

if (ps[i].bt\_remaining > 0) {

queue[rear++] = i;

visited[i] = true;

break;

}

}

}

}

max\_completion\_time = INT\_MIN;

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

max\_completion\_time = findmax(max\_completion\_time, ps[i].ct);

length\_cycle = max\_completion\_time - ps[0].at;

cpu\_utilization = (float)(length\_cycle - total\_idle\_time) / length\_cycle;

qsort((void \*)ps, n, sizeof(struct process\_struct), compare\_pid);

printf("\nProcess No.\tAT\tBT\tStart Time\tCT\tTAT\tWT\tRT\n");

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

printf("%d\t\t%d\t%d\t%d\t\t%d\t%d\t%d\t%d\n", ps[i].pid, ps[i].at, ps[i].bt, ps[i].start\_time, ps[i].ct, ps[i].tat, ps[i].wt, ps[i].rt);

}

printf("\nAverage Turn Around time= %.2f", (float)sum\_tat / n);

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

printf("\nAverage Response Time= %.2f", (float)sum\_rt / n);

printf("\nThroughput= %.2f", n / (float)length\_cycle);

printf("\nCPU Utilization(Percentage)= %.2f", cpu\_utilization \* 100);

// Print Gantt Chart

printf("\n\nGantt Chart:\n");

for (int i = 0; i < gc\_index; i++) {

printf("| P%d ", gantt\_chart[i]);

}

printf("|\n");

return 0;

}

**OUTPUT:**

**Question 2: Write a program to implement Priority Pre-emptive CPU Scheduling.**

**CODE:**

//Preemptive Priority

#include<stdio.h>

#include<stdbool.h>

struct process\_struct {

int at;

int bt;

int priority;

int ct, wt, tat, rt, start\_time;

} ps[100];

int main() {

int n;

bool is\_completed[100] = { false };

int bt\_remaining[100];

int current\_time = 0, completed = 0;

int gantt\_chart[1000], gc\_index = 0; // Array to store Gantt chart (assuming max 1000 time units)

// Input number of processes

scanf("%d", &n);

float sum\_tat = 0, sum\_wt = 0, sum\_rt = 0;

int i;

// Input arrival time, burst time, and priority

printf("\nEnter Process Arrival Time\n");

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

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

}

printf("\nEnter Process Burst Time\n");

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

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

bt\_remaining[i] = ps[i].bt; // Initialize remaining burst time

}

printf("\nEnter Priority\n");

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

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

}

while (completed != n) {

// Find the process with the highest priority that has arrived

int max\_index = -1;

int maximum = -1;

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

if (ps[i].at <= current\_time && is\_completed[i] == false) {

if (ps[i].priority > maximum) {

maximum = ps[i].priority;

max\_index = i;

}

if (ps[i].priority == maximum) {

if (ps[i].at < ps[max\_index].at) {

maximum = ps[i].priority;

max\_index = i;

}

}

}

}

if (max\_index == -1) {

current\_time++;

} else {

// If process starts for the first time, record start time

if (bt\_remaining[max\_index] == ps[max\_index].bt)

ps[max\_index].start\_time = current\_time;

// Execute the process for one unit of time

bt\_remaining[max\_index]--;

gantt\_chart[gc\_index++] = max\_index; // Store the process ID in Gantt chart

current\_time++;

// If process is completed

if (bt\_remaining[max\_index] == 0) {

ps[max\_index].ct = current\_time;

ps[max\_index].tat = ps[max\_index].ct - ps[max\_index].at;

ps[max\_index].wt = ps[max\_index].tat - ps[max\_index].bt;

ps[max\_index].rt = ps[max\_index].start\_time - ps[max\_index].at;

sum\_tat += ps[max\_index].tat;

sum\_wt += ps[max\_index].wt;

sum\_rt += ps[max\_index].rt;

completed++;

is\_completed[max\_index] = true;

}

}

}

// Print completion times

printf("\nCompletion times:\n");

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

printf("%d ", ps[i].ct);

}

// Print averages

printf("\nAverage Turn Around Time: %.2f", sum\_tat / n);

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

printf("\nAverage Response Time: %.2f", sum\_rt / n);

// Print Gantt Chart

printf("\n\nGantt Chart:\n");

for (int i = 0; i < gc\_index; i++) {

printf("| P%d ", gantt\_chart[i] + 1); // +1 for 1-based process numbering

}

printf("|\n");

return 0;

}

**OUTPUT:**