**LIST OF PROGRAMS (DAY 1)**

**S HARIHARAN (192211110)**

**1. Create a new process by invoking the appropriate system call. Get the process**

**identifier of the currently running process and its respective parent using system calls**

**and display the same using a C program.**

#include <stdio.h>

#include <unistd.h>

int main() {

printf("Process ID:%d\n",getpid());

printf("Parent Process ID: %d\n",getpid());

return 0;

}

**2. Identify the system calls to copy the content of one file to another and illustrate the same using a C program.**

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#define BUFFER\_SIZE 4096

int main() {

const char \*sourcefile ="C:\\Users\\ecute\\OneDrive\\Desktop\\aj.txt";

const char \*destination\_file ="C:\\Users\\ecute\\OneDrive\\Desktop\\ajay.txt";

// Open source file for reading

int source\_fd = open(sourcefile, O\_RDONLY);

if (source\_fd == -1) {

perror("Error opening source file");

return 1;

}

// Create or open destination file for writing

int dest\_fd = open(destination\_file, O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

if (dest\_fd == -1) {

perror("Error opening destination file");

close(source\_fd);

return 1;

}

char buffer[BUFFER\_SIZE];

ssize\_t bytesRead, bytesWritten;

// Read from source file and write to destination file

while ((bytesRead = read(source\_fd, buffer, BUFFER\_SIZE)) > 0) {

bytesWritten = write(dest\_fd, buffer, bytesRead);

if (bytesWritten != bytesRead) {

perror("Error writing to destination file");

close(source\_fd);

close(dest\_fd);

return 1;

}

}

// Check for read error

if (bytesRead == -1) {

perror("Error reading from source file");

close(source\_fd);

close(dest\_fd);

return 1;

}

// Close file descriptors

close(source\_fd);

close(dest\_fd);

printf("File copied successfully.\n");

return 0;

}

**3.Design a CPU scheduling program with C using First Come First Served technique with the following considerations.**

#include <stdio.h>

// Structure to represent a process

struct Process {

int process\_id; // Process ID

int arrival\_time; // Arrival time

int burst\_time; // Burst time

};

// Function to execute processes using FCFS scheduling

void executeFCFS(struct Process processes[], int n) {

int total\_waiting\_time = 0; // Total waiting time

int total\_turnaround\_time = 0; // Total turnaround time

int current\_time = 0; // Current time

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

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

// Calculate waiting time for the current process

int waiting\_time = current\_time - processes[i].arrival\_time;

if (waiting\_time < 0) {

// The process has not arrived yet, so the waiting time is 0.

waiting\_time = 0;

}

// Calculate turnaround time for the current process

int turnaround\_time = waiting\_time + processes[i].burst\_time;

// Update total waiting and turnaround times

total\_waiting\_time += waiting\_time;

total\_turnaround\_time += turnaround\_time;

// Print process details

printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].arrival\_time, processes[i].burst\_time, waiting\_time, turnaround\_time);

// Move the current time forward

current\_time += processes[i].burst\_time;

}

// Calculate and print average waiting and turnaround times

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

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

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

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

}

int main() {

int n; // Number of processes

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

scanf("%d", &n);

struct Process processes[n]; // Array to store processes

// Input process details

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

processes[i].process\_id = i + 1;

processes[i].arrival\_time = 0; // All processes are activated at time 0.

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

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

}

// Execute processes using FCFS scheduling

executeFCFS(processes, n);

return 0;

}

**4.Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.**

#include <stdio.h>

int main() {

int bt[20], p[20], wt[20], tat[20], i, j, n, total = 0, pos, temp;

float avg\_wt, avg\_tat;

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

scanf("%d", &n);

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

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

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

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

p[i] = i + 1;

}

// Sorting based on Burst Time using Selection Sort

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

pos = i;

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

if (bt[j] < bt[pos])

pos = j;

}

// Swapping Burst Time and Process ID

temp = bt[i];

bt[i] = bt[pos];

bt[pos] = temp;

temp = p[i];

p[i] = p[pos];

p[pos] = temp;

}

wt[0] = 0;

// Calculating Waiting Time and Total Waiting Time

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

wt[i] = 0;

for (j = 0; j < i; j++)

wt[i] += bt[j];

total += wt[i];

}

avg\_wt = (float)total / n;

total = 0;

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

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

tat[i] = bt[i] + wt[i];

total += tat[i];

printf("%d\t%d\t\t%d\t\t%d\n", p[i], bt[i], wt[i], tat[i]);

}

avg\_tat = (float)total / n;

printf("\nAverage Waiting Time = %f", avg\_wt);

printf("\nAverage Turnaround Time = %f\n", avg\_tat);

return 0;

}

**5. Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.**

#include<stdio.h>

struct priority\_scheduling {

char process\_name;

int burst\_time;

int waiting\_time;

int turn\_around\_time;

int priority;

};

int main() {

int number\_of\_process;

int total = 0;

struct priority\_scheduling temp\_process;

int ASCII\_number = 65;

int position;

float average\_waiting\_time;

float average\_turnaround\_time;

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

scanf("%d", & number\_of\_process);

struct priority\_scheduling process[number\_of\_process];

printf("\nPlease Enter the Burst Time and Priority of each process:\n");

for (int i = 0; i < number\_of\_process; i++) {

process[i].process\_name = (char) ASCII\_number;

printf("\nEnter the details of the process %c \n", process[i].process\_name);

printf("Enter the burst time: ");

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

printf("Enter the priority: ");

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

ASCII\_number++;

}

for (int i = 0; i < number\_of\_process; i++) {

position = i;

for (int j = i + 1; j < number\_of\_process; j++) {

if (process[j].priority > process[position].priority)

position = j;

}

temp\_process = process[i];

process[i] = process[position];

process[position] = temp\_process;

}

process[0].waiting\_time = 0;

for (int i = 1; i < number\_of\_process; i++) {

process[i].waiting\_time = 0;

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

process[i].waiting\_time += process[j].burst\_time;

}

total += process[i].waiting\_time;

}

average\_waiting\_time = (float) total / (float) number\_of\_process;

total = 0;

printf("\n\nProcess\_name \t Burst Time \t Waiting Time \t Turnaround Time\n");

printf("------------------------------------------------------------\n");

for (int i = 0; i < number\_of\_process; i++) {

process[i].turn\_around\_time = process[i].burst\_time + process[i].waiting\_time;

total += process[i].turn\_around\_time;

printf("\t %c \t\t %d \t\t %d \t\t %d", process[i].process\_name, process[i].burst\_time, process[i].waiting\_time, process[i].turn\_around\_time);

printf("\n-----------------------------------------------------------\n");

}

average\_turnaround\_time = (float) total / (float) number\_of\_process;

printf("\n\n Average Waiting Time : %f", average\_waiting\_time);

printf("\n Average Turnaround Time: %f\n", average\_turnaround\_time);

return 0;

}

**6. Construct a C program to implement pre-emptive priority scheduling algorithm.**

#include<stdio.h>

#include<conio.h>

int main()

{

int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];

float avg\_wt, avg\_tat;

printf(" Total number of process in the system: ");

scanf("%d", &NOP);

y = NOP;

for(i=0; i<NOP; i++)

{

printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);

printf(" Arrival time is: \t");

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

printf(" \nBurst time is: \t");

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

temp[i] = bt[i];

}

printf("Enter the Time Quantum for the process: \t");

scanf("%d", &quant);

printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0)

{

sum = sum + temp[i];

temp[i] = 0;

count=1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - quant;

sum = sum + quant;

}

if(temp[i]==0 && count==1)

{

y--;

printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

wt = wt+sum-at[i]-bt[i];

tat = tat+sum-at[i];

count =0;

}

if(i==NOP-1)

{

i=0;

}

else if(at[i+1]<=sum)

{

i++;

}

else

{

i=0;

}

}

avg\_wt = wt \* 1.0/NOP;

avg\_tat = tat \* 1.0/NOP;

printf("\n Average Turn Around Time: \t%f", avg\_wt);

printf("\n Average Waiting Time: \t%f", avg\_tat);

getch();

}

**7. Construct a C program to implement non-preemptive SJF algorithm.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#define SHM\_SIZE 1024

int main() {

pid\_t pid;

int shmid;

void \*shared\_memory;

char \*data;

// Create shared memory segment

shmid = shmget(IPC\_PRIVATE, SHM\_SIZE, IPC\_CREAT | 0666);

if (shmid == -1) {

perror("shmget");

exit(EXIT\_FAILURE);

}

// Attach the shared memory segment to the address space of the process

shared\_memory = shmat(shmid, NULL, 0);

if (shared\_memory == (void \*)-1) {

perror("shmat");

exit(EXIT\_FAILURE);

}

// Fork a child process

pid = fork();

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

// Child process (Reader)

data = (char \*)shared\_memory;

printf("Reader Process (PID %d) reading data from shared memory: %s\n", getpid(), data);

} else {

// Parent process (Writer)

data = (char \*)shared\_memory;

printf("Writer Process (PID %d) writing data to shared memory\n", getpid());

strcpy(data, "Hello, Shared Memory!");

// Wait for the child process to complete

wait(NULL);

// Detach and remove the shared memory segment

shmdt(shared\_memory);

shmctl(shmid, IPC\_RMID, NULL);

}

return 0;

}

**8. Construct a C program to simulate Round Robin scheduling algorithm with C.**

#include <stdio.h>

#define MAX\_PROCESS 10

struct Process {

int processID;

int arrivalTime;

int burstTime;

int remainingTime;

};

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

int time = 0;

int remainingProcesses = n;

while (remainingProcesses > 0) {

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

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

int executeTime = (processes[i].remainingTime < quantum) ? processes[i].remainingTime : quantum;

processes[i].remainingTime -= executeTime;

printf("Executing Process %d for %d units. Remaining time: %d\n", processes[i].processID, executeTime, processes[i].remainingTime);

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

remainingProcesses--;

printf("Process %d completed.\n", processes[i].processID);

}

time += executeTime;

}

}

}

}

int main() {

int n, quantum;

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

scanf("%d", &n);

struct Process processes[MAX\_PROCESS];

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

processes[i].processID = i + 1;

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

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

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

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

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

}

printf("Enter the time quantum for Round Robin scheduling: ");

scanf("%d", &quantum);

roundRobinScheduling(processes, n, quantum);

return 0;

}