Description

Unix offers a number of systems calls to handle a directory the following are most commonly used system calls.

System calls used

i. opendir()
open a directory

Syntax:-
DIR *opendir(const char *dirname);

opendir() takes dirname as the path name and returns a pointer to a DIR structure on error returns NULL.

ii. read dir()
Read a directory.

Synatx:
Struct dirent * readdir (DIR *dp);

A directory maintains the inode number and filename for every file its told this function returns a pointer to a dirent structure consisting of inode number and filename. 'dirent' structure is defined in <dirent.h> to provide at least two members :-inode number and directory name.

```
Struct dirent
{
  into_t d_into, // directory inode number
char d_name[ ]; // directory name
}
```

Date :
Exp. No.

Exp. Title Write the program to implement the system call opendir(), readdir(), closedir().

Page No.

ALGORITHM:

Step 1: Start the program.

Step 2: In the main function pass the argument.

Step 3: Create structure as stat buff and the variables as integers.

Step 4: Open the directory.

Step 5: Read the contents of the directory (filename).

Step 6: Display the contents of the directory.

Step 7: Close the directory.

Step 8: Stop the program.

Description

Unix offers a number of systems calls to handle a directory the following are most commonly used system calls.

System calls used

i. Opendir( )
open a directory

Syntax:-
DIR *opendir (const char * dirname);

opendir() takes dirname as the path name and returns a pointer to a DIR structure on error returns NULL.

ii. read dir( )
Read a directory.

Syntax:
Struct dirent * readdir (DIR * dp);

A directory maintains the inode number and filename for every file its told this function returns a pointer to a dirent structure consisting of inode number and filename. 'dirent' structure is defined in <dirent.h> to provide at least two members :-inode number and directory name.

```
Struct dirent
{
  into_t d-into , // directory inode number
char d-name[ ]; // directory name
}
```

Date :
Exp. No.

Exp. Title Write the program to implement the system call opendir(), readdir(), closedir().

Page No.

ALGORITHM:

Step 1: Start the program.

Step 2: In the main function pass the argument.

Step 3: Create structure as dirent buff and the variables as integer.

Step 4: Open the directory.

Step 5: Read the contents of the directory (filename).

Step 6: Display the contents of the directory.

Step 7: Close the directory.

Step 8: Stop the program.

::: closedir():
Close a directory

Syntax: int closedir (DIR* dp);
closes a directory pointed by dp. It returns 0 on success and -1 on error.

Date :	Exp. Title	Page No.
Exp. No.		

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <errno.h>
#include <string.h>

void read_file(const char* filepath)
{
    FILE *file = fopen(filepath, "r");
    if (file == NULL)
    {
        perror("Failed to open file");
        return;
    }
    char line[256];
    while (fgets(line, sizeof(line), file) != NULL)
    {
        printf("%s", line);
    }
    fclose(file);
}
int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <directory_name>\n", argv[0]);
        return EXIT_FAILURE;
    }
    DIR *dirp = opendir(argv[1]);
    if (dirp == NULL){
```

output:

File: Dev.c

contents of /Dev.c

```
#include <stdio.h>
void main()
{
printf ("Hello");
}
```

Date :

Exp. No.

Exp. Title

Page No.

```
perror ("Failed to open directory");
return EXIT_FAILURE;
}
struct dirent *direntp;
while (direntp = readdir(dirp)) != NULL){
if (strcmp (direntp->d_name, ".") != 0 && strcmp (direntp->
    d_name, "..") != 0)
{
if (strcmp
printf("File:%s \n", direntp->d_name);
if (direntp->d_type == DT_REG){
char filepath[1024];
snprintf(filepath, sizeof(filepath), "%s%s", argv[1], direntp->
d_name);
printf("contents of %s: \n", filepath);
read_file(filepath);
}
printf("\n");
}
if(closedir(dirp) == -1)
{
perror ("Failed to close directory");
return EXIT_FAILURE;
}
return EXIT_SUCCESS;
}
}
```

Date :

Exp. No.

Exp. Title

Page No.

ALGORITHM: Shortest Running Time Implementation.

Purpose: To represent the working of the shortest remaining time algorithm.

Input: No of process; Burst time of each process & quantum time.

Output: Average waiting & turnaround time.

Date :

Exp. No.

Exp. Title

Page No.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/type.h>
#include <sys/wait.h>

int main()
{
 int pid[15];
 int bt[15];
 int n;
 printf("Enter the number of processes");
 scanf("%d", &n);
 printf("Enter process id of all the processes");
 for(int i=0; i<n; i++)
 {
  scanf("%d", &pid[i]);
 }
 printf("Enter burst time of all the processes);
 for(int i=0; i<n; i++)
 {
  scanf("%d", &bt[i]);
 }
 int i, wt[n];
 wt[0] = 0;
 for(i=1; i<n; i++)
 {
   wt[i] = bt[i-1] + wt[i-1];
 }
printf("process ID Burst Time waiting Time
        Turnaround Time \n");
```

Output: Enter the number of process. 5
Enter process id of all the process: 1 2 3 4 5
Enter burst time of all the process: 10 8 5 7 9

ProcessID	Burst time	waiting time	TurnAround time
1	10	0	10
2	8	10	18
3	5	18	23
4	7	23	30
5	9	30	39

Avg. waiting time = 16.200001
Avg. turnaround time = 24.000000

Date :
Exp. No.
Exp. Title
Page No.

```
    float wot = 0.0;
    float tat = 0.0;
    for (i=0; i<n; i++)
    {
    printf("%d\t\t", pid[i]);
    printf("%d\t\t", bt[i]);
    printf("%d\t\t", wt[i]);
    // Calculating & printing turnaround time of each process
    printf("%d\t\t", bt[i] + wt[i]);
    printf("\n");
    twt += wt[i];
    tat += (wt[i] + bt[i]);
    }
    float att, awt;
    awt = twt/n;
    att = tat/n;
    printf("Avg. waiting time = %f\n", awt);
    printf("Avg. turnaround time = %f", att);
}
```

Description :-

- The CPU is assigned to the process that has the smallest next CPU host.
- If two processes have the same length CPU burst. FCFS scheduling is used to break the file.
- For long-term scheduling in a batch system, we can use the process time limit specified by the user as the length.
- SRCF can't be implemented at the level of short-term scheduling because there is no way to know the length of the next CPU burst.
- If the new process has a shorter next CPU burst than what is left of the executing process that process is preempted.

Date :

Exp. No.

Exp. Title

Page No.

Algorithm:

Step 1: Define a structure to hold process details.

Step 2: Input the number of processes and their details.

Step 3: Sort Processes by arrival time

Step 4: Calculate completion time using the SRTF algorithm. For each time unit find the process with the shorted remaining time and execute it.

Step 6: Calculate waiting times for each process.

Step 7: Display the process details in a table.

Step 8: Calculate and display average waiting and turnaround time.

Date :

Exp. No.

Exp. Title

Page No.

```
# include <stdio.h>
# define MAX 100
struct process
{
 int id;
 int arrival;
 int burst;
 int remaining;
 int completion;
 int turn_around;
 int waiting;
};
void findwaitingTime (struct process proc[], int n)
{
 proc [0].waiting = 0;
 for (int i = k; i<n; i++)
 {
  pro[i]. waiting = proc [i-1]. completion - proc[i].arrival;
 }
}
 void findTurnaroundTime (struct process proc[],
                          int n)
{
 for(int i=0; i<n; i++)
 {
 proc [i].turn_around = proc [i]. waiting + proc [i]. burst;
}
}
```

ate :

xp. No.

Exp. Title

Page No.

```
Void findCompletionTime (struct process proc[], int n)
{
int currentTime = 0;
while (1)
{
int minRemainingTime = 999;
int shortestprocessIndex = -1;
int remaining processes = 0;

for (int i = 0; i<n; i++)
{
 if (Proc[i].arrival <= currentTime && process[i].
    remaining > 0)
 {
 remainingprocess ++;
if (proc[i].remaining < minRemainingtime)
{
 minRemainingTime = proc[i].remaing;
 shortestprocessIndex = i;
 }
}
}
 if (remaining process == 0)
{
   break;
}
proc [shorestprocessIndex].remaining --;
 current time ++;
if (proc [short processIndex].remaining == 0)
{
  Proc [shortest processIndex].completion = currenttime;
```

Date :

Exp. No.

Exp. Title

Page No.

```
}
}
}
void calculateAverageTimes(struct process
proc[], int n)
{
int totalTurnAroundTime = 0, totalwaitingTime = 0;
for (int i=0; i<n; i++)
{
total.TurnAroundTime t = proc[i] trun-around;
total waiting time t = proc[i]. waiting;
}
printf("\n Average waiting time = %.2f ", (float)
totalwaiting time \n);
printf("\n Average Turnaround Time = %.2 f", (float)
total TurnaroundTime \n);
}
int main()
{
int n;
printf("Enter the number of processes:");
scanf("%d", &n);
struct process proc[MAX];
for(int i=0; i<n; i++)
{
printf("\n Enter details for process %d \n", i+1);
proc[i]. id = i+1;
printf("Arrival time:");
scanf("%d", &proc[i]. arrival);
printf("Burst Time:");
scanf("%d", &proc[i]. burst);
```

Date :

Exp. No.

Exp. Title

Page No.

```
    proc[i].remaining = proc[i].burst;
}
for(int i=0; i<n-1; i++)
{
for (int j=i+1; j<n; j++)
{
if (proc[i].arrival > proc[j].arrival)
{
    struct process temp = proc[i];
    proc[i] = proc[j];
    proc[j] = temp;
}
}
}
find completionTime(proc,n);
find waitingTime(proc, n);
find TurnAroundTime(proc, n);
printf("\n Process ID| Arrival Time | Burst Time|
        Completion Time| Waiting Time| Turnaround Time\n");
for(int i=0; i<n; i++)
{
printf("%d \t %d \t %d \t %d \t %d \t %d \t", proc[i].id,
  proc[i].arrival, proc[i].burst, proc[i].completion,
  proc[i].waiting, proc[i].turn-around);
}
Calculate AverageTime(proc, n);
 return 0;
}
```

Output

Description:-

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum.
- once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

Date :
Exp. No.
Exp. Title
Page No.

Algorithm:

Step 1: Start the program.

Step 2: Initialize all the structure elements.

Step 3: Receive inputs from the user to fill process id, burst time and arrival time.

Step 4: Calculate the waiting time for all the process id.

i) The waiting time for first instance of a process is calculated as: a[i].wait time = count + a[i].arrival.

ii) The waiting time for the rest of the instances of the process is calculated as

a) If the time quantum is greater then the remaining burst time then waiting time is calculate as a[i].waiting = count + tq

b) Else if the time quantum is greater than the remaining burst time then waiting time is calculated as; a[i].wait time = count - remaining burst time.

Step 5: Calculate the average waiting time and average Arounround time.

Step 6: Print the results of the step4.

Step 7: Stop the program.

Date :

Exp. No.

Exp. Title

Page No.

```
# include <stdio.h>
# define MAX 10
 struct process
 {
   int pid;
   int bt;
   int art;
   int wt;
   int tat;
   int rem_bt;
   int completed;
};
   void findwaiting time (struct process proc[], int n,
        int quantum)
{
  int time = 0;
  int remaining_processes = n;
  for(int i=0; i<n; i++)
  {
   proc[i].rem_bt = proc[i].bt;
   proc[i].completed = 0;
}
  while (remaining_process > 0)
  {
  for(int i=0; i<n; i++)
  {
  if (proc[i].art <= time && proc[i].rem_bt > 0
      && proc[i].completed == 0)
{
  if (proc[i].rem_bt > quantum)
```

Date :

Exp. No.

Exp. Title

Page No.

```
    {
      time += quantum;
      proc[i].rem_bt -= quantum;
    }
    else
    {
      time += proc[i].rem_bt;
      proc[i].wt = time - proc[i].bt - proc[i].art;
      proc[i].tat = proc[i].wt + proc[i].bt;
      proc[i].rem_bt = 0;
      proc[i].completed = 1;
      remaining_processes--;
    }
  }
 }
}
void calculateAverageTimes(struct process proc[], int n)
{
 int total_wt = 0, total_tat = 0;
 for (int i=0; i<n; i++)
 {
  total_wt += proc[i].wt;
  total_tat += proc[i].tat;
 }
 printf("\n Average waiting Time : %.2f", (float)
        total_wt / n);
 printf("\n Average Turnaround Time: %.2f", (float)
        total_tat / n);
}
```

Output:

```
Enter number of processes: 4
Enter Burst Time and Arrival Time for process 1: 8 0
Enter Burst Time and Arrival Time for process 2: 4 1
Enter Burst Time and Arrival Time for process 3: 9 2
Enter Burst Time and Arrival Time for process 4: 5 3

Enter Time Quantum: 2
```

Process ID	Arrival Time	Burst Time	Waiting Time	Turnaround Time
1	0	8	15	23
2	1	4	17	11
3	2	9	15	24
4	3	5	13	18

```
Average Waiting Time: 12.50
Average Turnaround Time: 19.00
```

Date :
Exp. No.
Exp. Title
Page No.

```
int main()
{
int n, quantum;
struct process proc[MAX];
printf("Enter number of processes:");
scanf("%d", &n);
for (int i=0; i<n; i++)
{
printf("Enter Burst Time and Arrival Time for
    process %d:", i+1);
scanf("%d %d", &proc[i].bt, &proc[i].art);
  proc[i].pid = i+1;
}
printf("Enter Time Quantum:");
scanf("%d", &quantum);
findwaitingTime(proc, n, quantum);
printf("\n process ID | Arrival Time | Burst Time |
 waiting Time | Turnaround Time \n");
for (int i=0; i<n; i++)
{
printf("%d \t \t %d \t \t %d \t \t %d \t \t %d \t \t \n", proc[i].
    pid, proc[i].art, proc[i].bt, proc[i].wt,
 proc[i].tat);
}
calculateAverageTimes(proc,n);
  return 0;
}
```