Github link : https://github.com/pranjalbansal787/CSE316

Project code :16

Name : Pranjal Bansal

Registration : 11809269

Section : K18NZ   Roll No 44

Design a scheduler that can schedule the processes arriving system at periodical intervals. Every process is assigned with a fixed time slice t milliseconds. If it is not able to complete its execution within the assigned time quantum, then automated timer generates an interrupt. The scheduler will select the next process in the queue and dispatcher dispatches the process to processor for execution. Compute the total time for which processes were in the queue waiting for the processor. Take the input for CPU burst, arrival time and time quantum from the user.

Description:

Design a scheduler that can schedule the processes arriving system at periodical intervals. Every process is assigned with a fixed time slice t milliseconds. If it is not able to complete its execution within the assigned time quantum, then automated timer generates an interrupt. The scheduler will select the next process in the queue and Dispatcher dispatches the process to processor for execution. Compute the total time for which processes were in the queue waiting for the processor. Take the input for CPU burst, arrival time and time quantum from the user. The Algorithm focuses on Time Sharing. In this algorithm, every process gets executed in a cyclic way. A certain time slice is defined in the system which is called time quantum. Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will terminate else the process will go back to the ready queue and waits for the next turn to complete the execution.

Explanation:

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.

Example: Assume there are 5 processes with process ID and burst time given below PID Burst Time P1 6 P2 5 P3 2 P4 3 P5 7 – Time quantum: 2 – Assume that all process arrives at 0. Now, we will calculate average waiting time for these processes to complete. Solution – We can represent execution of above processes using GANTT chart as shown below –

Explanation: – First p1 process is picked from the ready queue and executes for 2 per unit time (time slice = 2). If arrival time is not available, it behaves like FCFS with time slice. – After P2 is executed for 2 per unit time, P3 is picked up from the ready queue. Since P3 burst time is 2 so it will finish the process execution at once. – Like P1 & P2 process execution, P4 and p5 will execute 2 time slices and then again it will start from P1 same as above. Waiting time = Turn Around Time – Burst Time P1 = 19 – 6 = 13 P2 = 20 – 5 = 15 P3 = 6 – 2 = 4 P4 = 15 – 3 = 12 P5 = 23 – 7 = 16 Average waiting time = (13+15+4+12+16) / 5 = 12

Algorithm:

1.Create an array arrival_time[], burst_time[] to keep track of arrival and bust time of processes.

2.Create another array temp[] to store waiting times of processes temporarily in between execution.

3.Initialize time: total = 0, counter = 0, wait time = 0, turnaround time = 0.

4.Ask user for no of process and store it in limit.

5.Repetitively ask user to give input for - arrival time, bust time up to limit.

6.Ask user to enter time quantum and store it into time_quantum.

7.Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet. a. If temp[i] <= time_quantum && temp[i] > 0 (i) total = total + temp[i]; (ii) temp[i] = 0; (iii) counter = 1; b. Else if temp[i] > 0 (i) temp[i] = temp[i] - time_quantum; (ii) total = total + time_quantum; c. If temp[i] == 0 && counter == 1 (i) x--; (ii) print burst_time[i], total, arrival_time[i], total - arrival_time[i] - burst_time[i]); (iii) wait_time = wait_time + total - arrival_time[i] - burst_time[i]; (iv) turnaround_time = turnaround_time + total - arrival_time[i]; (v) counter = 0; d. If i == limit – 1 (i) i = 0; e. Else if arrival_time[i + 1] <= total (i) i++; c. Else (i) i = 0;

8.Print total waiting time - wait_time, average waiting time - wait_time/limit, average turnaround time - average_turnaround_time/limit.

Source Code:

```
#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>
```

```
void waiting(int n,int a[10], int b[10], int x[10]);
int main()
{
    int n;
    printf("\t\t\t\t*************************************************");
    printf("\n\t\t\t\t*                              *");
    printf("\n\t\t\t\t*            WAITING TIME          *");
    printf("\n\t\t\t\t*                 by Pranjal bansal *");
    printf("\n\t\t\t\t*************************************************");



    int choice;
    printf("\n\nMain Menu\n\nPress:\n1. Process Scheduling\n2. About\n3. Exit");
    printf("\n>");
    scanf("%d",&choice);
    switch(choice){
    case 1:{
    process_again:
    printf("\n\nEnter the number of Processes:\n");
    printf(">");
    scanf("%d",&n);
    if(n<=0){
        printf("\nProcess cannot be 0 or less, Try Again...\n");
        goto process_again;
    }
```

```
int a[n],b[n],x[n],i,j,k=0,p_no[n];

for(i=0;i<n;i++){
  printf("\nProcess : %d",i+1);
  p_no[i]=i+1;
  a_again:
  printf("\nEnter Arrival Time: ");
  scanf("%d",&a[i]);
  if(a[i]<0){
    printf("\nArrival Time cannot be less then 0, Try again...\n");
    goto a_again;
  }
  b_again:

  printf("Enter Burst Time: ");
  scanf("%d",&b[i]);
  if(b[i]<=0){

    printf("\nBurst Time cannot be 0 or less, Try again...\n");
    goto b_again;
  }
  x[i]=b[i];
}
waiting(n,a,b,x);
break;
```

```
 }

 case 2:{

   M_again:

   system("clear");

   printf("\t\t\t\t************************************************");

   printf("\n\t\t\t\t*                                    *");

   printf("\n\t\t\t\t*              WAITING TIME          *");

   printf("\n\t\t\t\t*                        by Pranjal Bansal*");

   printf("\n\t\t\t\t************************************************");



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

   char buff[565]="Design a scheduler that can schedule the processes arriving system at
periodical intervals. Every process is assigned with a fixed time slice t milliseconds. If it is not
able to complete its execution within the assigned time quantum, then automated timer
generates an interrupt. The scheduler will select the next process in the queue and dispatcher
dispatches the process to processor for execution. Compute the total time for which processes
were in the queue waiting for the processor. Take the input for CPU burst, arrival time and time
quantum from the user.";

   printf("%s",buff);

   printf("\n\nPress M to go back to Main Menu: ");

   char b;

   fflush(stdin);

   scanf("%s",&b);

   if(b=='m' || b=='M'){

     main();

   }

   else{
          goto M_again;
        }
    }
    case 3:{
        printf("\nThank you for using this Program.....\n");
        exit(0);
    }
    default:{
        printf("\nInvalid Selection....\n");
        break;
    }
    }
    return 0;
}

//Function to calculate the Waiting Time for the Processes
void waiting(int n,int a[10],int b[10],int x[10]){
int time_quantum,ccount[n],count=0,i;
 tq_again:
 printf("\nEnter Time Quantum: ");
 scanf("%d",&time_quantum);
 if(time_quantum<=0){
     printf("\nTime Quantum cannot be 0 or less, Try again...\n");
     goto tq_again;
 }
```

```
int loc=0,torun=1,time_slice=0,min=0,p_ctime=0;


//checking if processes are starting from zero or not

int minfind=0;

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

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

      if(a[i]<a[j]){

         minfind=a[i];

      }

   }

}


int zero_check=0;

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

   if(a[i]==0){

      break;

   }

   else{

      zero_check=minfind;

   }

}

time_slice=zero_check; //starting time of running queue


int
flag_to_check_two_same_burst_time=0,flag_to_check_two_same_arrival_time=0,loc2=0,loc3=0,
sjf_run=0;
printf("\n\nRunning...\n");
while(torun){ //running untill all process's burst time becomes zero
   min=9999;
   sjf_run=0;
   //checking two arrival time and selecting process wise.
   for(int i2=0;i2<n;i2++){
         for(int i3=i2+1;i3<n;i3++){
            if(a[i2]==a[i3] && (i2!=loc || i2==0) && b[i2]!=0){
               flag_to_check_two_same_arrival_time=1;
               loc3=i2;
               break;
            }
         }
         if(flag_to_check_two_same_arrival_time==1){
            break;
         }
      }

   //checking two burst time same then select the one came first.
   for(int i2=0;i2<n;i2++){
         for(int i3=i2+1;i3<n;i3++){
            if(b[i2]==b[i3] && b[i2]!=0){
               flag_to_check_two_same_burst_time=1;
               loc2=i2;
            break;
          }
        }
        if(flag_to_check_two_same_burst_time==1){
          break;
        }
      }

//if arrival time and burst time are same then run according to the process
if(flag_to_check_two_same_burst_time==1  && flag_to_check_two_same_arrival_time==1){
          min=b[loc3];
          loc=loc3;
          flag_to_check_two_same_burst_time=0;
          flag_to_check_two_same_arrival_time=0;
          sjf_run=1;
}

//if only burst time is same then run according to the arrival time;
if(flag_to_check_two_same_burst_time==1){
          min=b[loc2];
          loc=loc2;
          flag_to_check_two_same_burst_time=0;
          sjf_run=1;
}
```

```
//if no arrival time or burst time are same then run sjf
if(sjf_run==0){
for(i=0;i<n;i++)
{ if((a[i]<=time_slice && b[i]<=min && b[i]>0)||min==0){
          min=b[i];
          loc=i;
   }
}
}

 b[loc]=b[loc]-time_quantum;
 if(b[loc]==0){
    count++;
    ccount[loc]=time_slice+time_quantum-p_ctime;
 }
 if(b[loc]<0){
    count++;
    ccount[loc]=time_slice+time_quantum+b[loc]-p_ctime;
    p_ctime=(-(b[loc]));
    b[loc]-=b[loc];
 }
 if(count==n){ //end the process
    torun=0;
 }
 sleep(1);
```

```
     printf("\n\n");
     printf("\nProcess\t\tArrival Time\t\tBurst Time\t\tRemaining Time");
     printf("\nP%d\t\t\t%d\t\t\t%d\t\t\t   %d",loc+1,a[loc],x[loc],b[loc]);
     printf("\n---------------------------------------------------------------");
    time_slice+=time_quantum;  //adding time quantum every time
}


int turn_arround_time[n];
for(i=0;i<n;i++){
    turn_arround_time[i]=ccount[i]-a[i];  //calculating Turn Arround Time
}

int waiting_time[n],total_wt=0;
for(i=0;i<n;i++){
    waiting_time[i]=turn_arround_time[i]-x[i];  //Calculating Waiting Time
    if(waiting_time[i]<0){
        waiting_time[i]=0;
    }
    total_wt+=waiting_time[i];  //Total Waiting Time
}


   printf("\n\n\n-------------\n");
   printf("Overall Result\n-------------\n");
```

```
  printf("Process\t\tArrival Time\t\tBurst Time\t\tCompletion Time\t\tWaiting Time");
  printf("\n-----------------------------------------------------------------------------");
  for(i=0;i<n;i++){
  printf("\nP%d\t\t\t%d\t\t\t%d\t\t\t%d\t\t\t%d",i+1,a[i],x[i],ccount[i],waiting_time[i]);
  }

 printf("\n\nTotal Waiting time = %d\n\n",total_wt);
}
```

Constraints:

If we have large number of processes and a process with very less burst time. According to round robin it will have to wait for very long time if burst time of other processes is large and its turn comes late.

If we have some processes running according to round robin and we have a process whose arrival time is after the completion of those processes, then that process will not execute. If we want that all the processes should execute successfully then the arrival time of other process must be less than or equal to the completion time of the running processes.

Test Cases:

If we select same arrival time for 2 processes. Expected Result: The process having less burst time should have less turnaround time. Actual Result:

Test case passed successfully.

If p1 process arrives at 2 and burst time=4 but p2 and p3 process arrives at 6.

Expected Result: Here only process p1 will execute because p2 and p3 are unable to arrive within the completion of process p1.

Actual Result:

Test case passed successfully.

Now if process p1 arrives at 0 and two other processes p2 and p3 arrives within the completion of process p1.

Expected Result:

All 3 process will execute.

Actual Result:

Test case passed successfully.

Conclusion: Hence from the test cases we conclude that if we want that all the processes should execute successfully then the arrival time of other process must be less than or equal to the completion time of the running processes