Student Name: Ambati Vishnu Teja Reddy

Student ID: 11811604

Email : Vishnutejareddyambati@gmail.com

GitHub Link: https://github.com/tejareddy944/os-project

Code:

```
In rou c#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 Vishnu Teja*");
    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 SMD Saif *");
    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,loc
3=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);
}
```

Round robin scheduling 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 pre-empted and other process executes for a given time period. Context switching is used to save states of pre-empted processes.

Description:

In round robin scheduling, all process will complete their execution according to quantum time of 4. Processes are assigned to a queue on entry in the system processes do not move between queues. This setup has the advantage of low scheduling overhead. The main idea is that to separate process with different CPU-burst characteristics. If a process uses two much CPU time, it will be moved to a lower priority queue. Similarly, a process that waits too long in a lower priority queue may be moved to a higher priority queue. This form of aging prevents starvation.

The following parameters are used in this solution:

- The number of queues.
- The scheduling algorithm for each queue.
- The method used to determine when to upgrade a process to a higher priority queue.
- The method used to determine when to demote a process to a lower priority queue.
- The method used to determine which queue a process will enter when that process needs Service.

In this we use most general CPU scheduling algorithms. It can be configured to match a specific system under design. It also requires some means of selecting values for all given parameters to get the best solution in easy manner.

1.Round Robin is the preemptive process scheduling algorithm.

2.Each process is provided a fix time to execute, it is called a quantum.

3.Once a process is executed for a given time period, it is preempted and other process executes for a given time period.

4.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

<u>Algorithm:</u>

Round Robin scheduling complexity calculation are done by given formulas:

- Arrival Time: Time at which the process arrives in the ready queue.
- Completion Time: Time at which process completes its execution.
- Burst Time: Time required by a process for CPU execution.
- Turn Around Time: Time Difference between Completion time and Arrival time.
- Turn Around Time = Completion Time – Arrival Time.
- Waiting Time (W.T): Time Difference between Turn Around time and Burst Time.
- Waiting Time = Turn Around time – Burst time

<u>Description (purpose of use):</u>

When a process is given the CPU, a timer is set for whatever value has been set for a time quantum. If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU. If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue. The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on. RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms. The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally. BUT, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are. In general, turnaround time is minimized if most processes finish their next CPU-burst within one- time quantum. Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.

<u>Code Snippet:</u>

I have not used any additional algorithm because, I have used round robin scheduling for doing solution of my given problem.

Description:

In this solution the following conditions are used:

- Minimum context switches.
- Maximum CPU utilization.
- Maximum throughput.
- Minimum turnaround time.
- Minimum waiting time.

Description:

Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority. Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind. The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behaviour as FCFS. If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

Description:

I have made the revision of solution for 5 times in GitHub and done the solution without any errors.

GitHub Link: https://github.com/tejareddy944/os-project