OPERATING SYSTEMS

CSE 2005

EPJ COMPONENT

REVIEW-3

SCHEDULING ALGORITHM

SUBMITTED BY:

KHYATI CHATURVEDI-19BCI0124

BHAVINI SINGH-19BCI0100

- **Video Drive link:**
  https://drive.google.com/file/d/1XlfTAhGEUfsJb6kj7qLgOvbuzuPZVYrT/view?usp=sharing

ABSTRACT

This project deals with the simulation of CPU scheduling algorithms with C. The following algorithms are simulated:
1.First Come First Serve (FCFS)
2.Shortest Job First
3.SRTF Algorithm
4.Round Robin
5.Our innovative algorithm
The metrics such as finishing time, waiting time, total time taken for the processes to complete, number of rounds, etc are calculated.

SOFTWARE ARCHITECTURE (C CODE) :

There are five files for the five scheduling algorithms. They are made into each functions. This function is imported to main function which has the menu. The menu has the choice of which of the algorithm it want to get executed. In each function the code for each algorithm is done. The code consists arrival time and burst time for each process. The output has the waiting time and turnaround time for each process. At the beginning any number of processes can be taken in account. The output of each process with all the algorithms can be found.

LITERATURE REVIEW

- *Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)*
- *In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.*
- *A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.*
- *The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.*

CPU-I/O Burst Cycle

- Almost all processes alternate between two states in a continuing cycle:
  - ✓ A CPU burst of performing calculations, and
  - ✓ An I/O burst, waiting for data transfer in or out of the system.

CPU Scheduler

- Whenever the CPU becomes idle, it is the job of the CPU Scheduler (a.k.a. the short-term scheduler) to select another process from the ready queue to run next.
- The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue. There are several alternatives to choose from, as well as numerous adjustable parameters for each algorithm

PRE-EMPTIVE SCHEDULING

- CPU scheduling decisions take place under one of four conditions:
  1. When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait( ) system call.
  2. When a process switches from the running state to the ready state, for example in response to an interrupt.
  3. When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait( ).
  4. When a process terminates.

- For conditions 1 and 4 there is no choice - A new process must be selected.
- For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.

- If scheduling takes place only under conditions 1 and 4, the system is said to be non-pre-emptive, or cooperative. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be pre-emptive.

- Windows used non-pre-emptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Macs used non-pre-emptive prior to OSX, and pre-emptive since then. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt.
- Note that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures.
- Pre-emption can also be a problem if the kernel is busy implementing a system call (e.g. updating critical kernel data structures) when the pre-emption occurs. Most modern UNIXs deal with this problem by making the process wait until the system call has either completed or blocked before allowing the pre-emption Unfortunately this solution is problematic for real-time systems, as real-time response can no longer be guaranteed.
- Some critical sections of code protect themselves from con currency problems by disabling interrupts before entering the critical section and re-enabling interrupts on exiting the section. Needless to say, this should only be done in rare situations, and only on very short pieces of code that will finish quickly, (usually just a few machine instructions.)

DISPATCHER

The dispatcher is the module that gives control of the CPU to the process selected by the scheduler. This function involves:

- Switching context.
- Switching to user mode.
- Jumping to the proper location in the newly loaded program.
- The dispatcher needs to be as fast as possible, as it is run on every context switch. The time consumed by the dispatcher is known as dispatch latency.

SCHEDULING CRITERIA

There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:

- CPU utilization - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

- Throughput - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
- Turnaround time - Time required for a particular process to complete, from submission time to completion. (Wall clock time.)
- Waiting time - How much time processes spend in the ready queue waiting their turn to get on the CPU.
    - ✓ (Load average - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who".)
- Response time - The time taken in an interactive program from the issuance of a command to the commence of a response to that command.

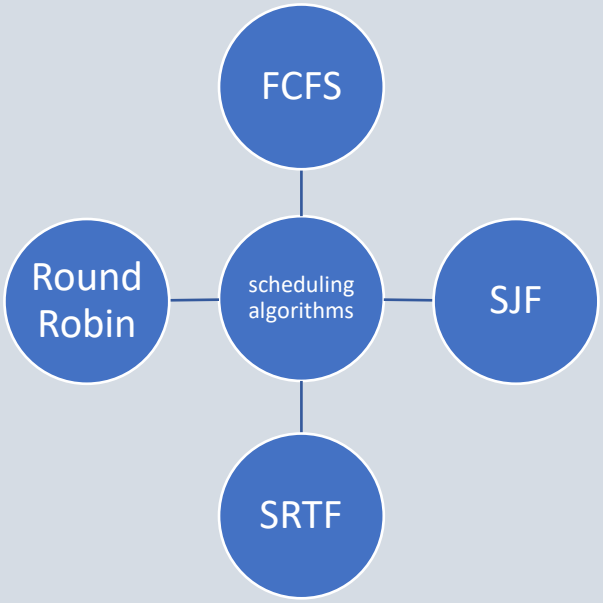
In general, one wants to optimize the average value of a criteria (Maximize CPU utilization and throughput, and minimize all the others.) However sometimes one wants to do something different, such as to minimize the maximum response time.

Sometimes it is most desirable to minimize the variance of a criteria than the actual value. I.e. users are more accepting of a consistent predictable system than an inconsistent one, even if it is a little bit slower.

SCHEDULING ALGORITHM

1. What is CPU Scheduling?

- The process which determines which process will be executed first and which one will be put on hold is known as CPU Scheduling.
- The responsibility to make sure a process enters the ready queue for execution and the CPU whenever sits idle is assigned a process.
- There are mainly five different types of scheduling algorithms:



2.TYPES OF SCHEDULING ALGORITHMS

- **FCFS SCHEDULING:**
  - First-Come-First-Served algorithm is the simplest scheduling algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue.
  - FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time.
  - The code for FCFS scheduling is simple to write and understand.
  - One of the major drawbacks of this scheme is that the average time is often quite long
- **SJF ALGORITHM**:
  - Shortest-Job-First (SJF) is a non-pre-emptive discipline in which waiting job (or process) with the smallest estimated run -time-to-completion is run next. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst.
  - The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal.
  - The SJF algorithm favours short jobs (or processors) at the expense of longer ones.

- The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available.

- **SRTF ALGORITHM**:

  - The SRT is the pre-emptive counterpart of SJF and useful in time-sharing environment.
  - In SRT scheduling, the process with the smallest estimated run-time to completion is run next, including new arrivals.
  - In SJF scheme, once a job begins executing, it run to completion.
  - In SJF scheme, a running process may be pre-empted by a new arrival process with shortest estimated run-time.
  - The algorithm SRT has higher overhead than its counterpart SJF.

  - The SRT must keep track of the elapsed time of the running process and must handle occasional pre-emptions.
  - In this scheme, arrival of small processes will run almost immediately. However, longer jobs have even longer mean waiting time

- **ROUND ROBIN SCHEDULING**:

  - One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR).
  - In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time called a time-slice or a quantum.
  - If a process does not complete before its CPU-time expires, the CPU is pre-empted and given to the next process waiting in a queue. The pre-empted process is then placed at the back of the ready list.
  - The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS.

3. THE PROBLEM:

1.First Come First Serve:

- *The process with less execution time suffers i.e. waiting time is often quite long.*
- *Favours CPU Bound process then I/O bound process.*
- *Here, first process will get the CPU first, other processes can get CPU only after the current process has finished its execution. Now, suppose the first process has large burst time, and other processes have less burst time, then the processes will have to wait more unnecessarily, this will result in more average waiting time, i.e., Convey effect.*
- *This effect results in lower CPU and device utilization.*
- *FCFS algorithm is particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.*

2.Shortest Job First:

- *SJF may cause starvation, if shorter processes keep coming. This problem is solved by aging.*
- *It cannot be implemented at the level of short-term CPU scheduling.*

3.Round Robin:

- *Setting the quantum too short, increases the overhead and lowers the CPU efficiency, but setting it too long may cause poor response to short processes.*
- *Average waiting time under the RR policy is often long.*

4.SRTF:

- *It cannot be implemented practically since burst time of the processes cannot be known in advance.*
- *It leads to starvation for processes with larger burst time.*
- *Priorities cannot be set for the processes.*
- *Processes with larger burst time have poor response time.*

4. Need for a new algorithm

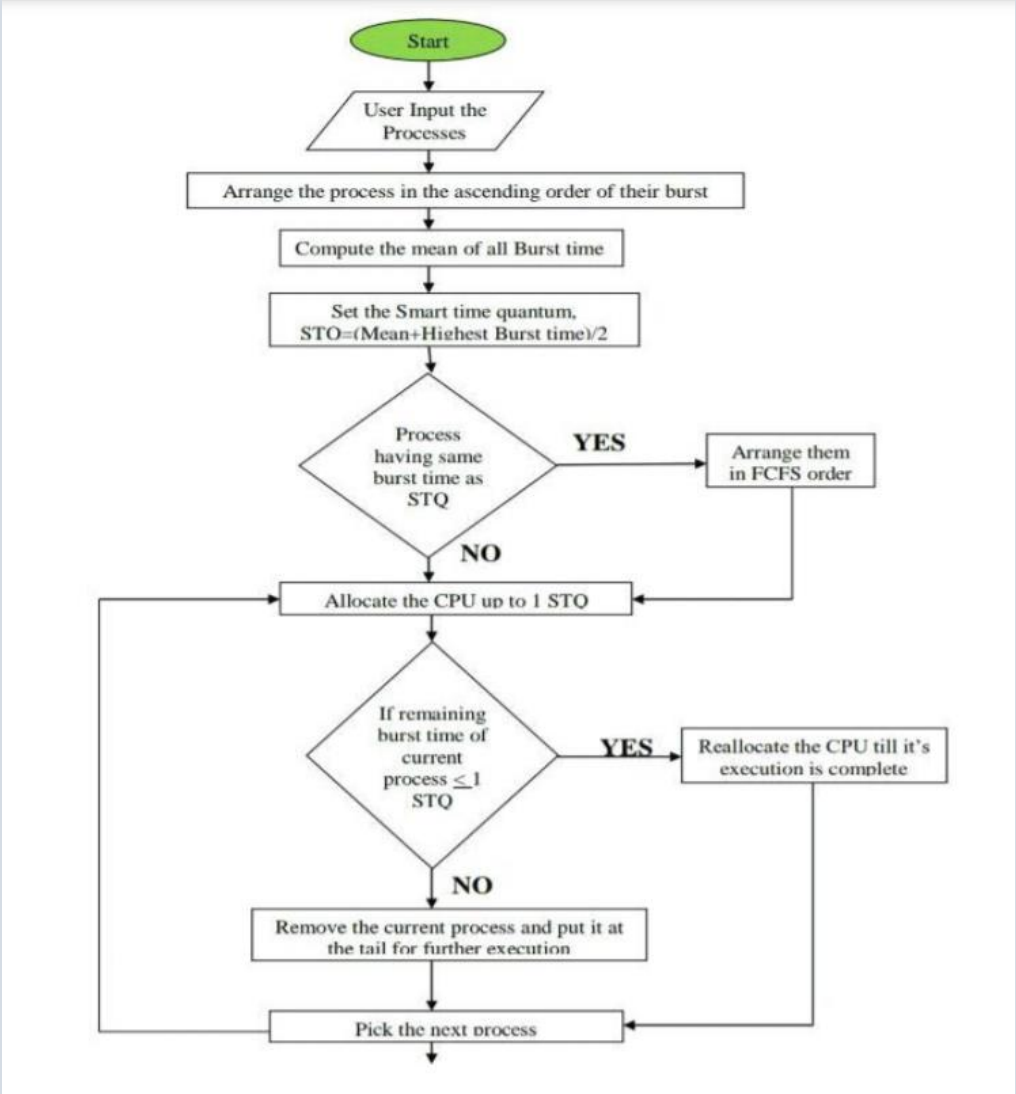
- Round Robin, considered as the most widely adopted CPU scheduling algorithm but it undergoes severe problems which are directly related to quantum size.
- If time quantum chosen is too large, the response time of the processes is considered too high.
- On the other hand, if this quantum is too small, it increases the overhead of the CPU.
- In this paper, after referring to a number of papers on this topic we have proposed a new algorithm , based on a new approach called dynamic-time-quantum.
- The idea of this approach is to make the operating systems adjusts the time quantum according to the burst time of the set of waiting processes in the ready queue.
- Based on the simulations and experiments, we show that the new proposed algorithm solves the fixed time quantum problem and increases the performance of Round Robin

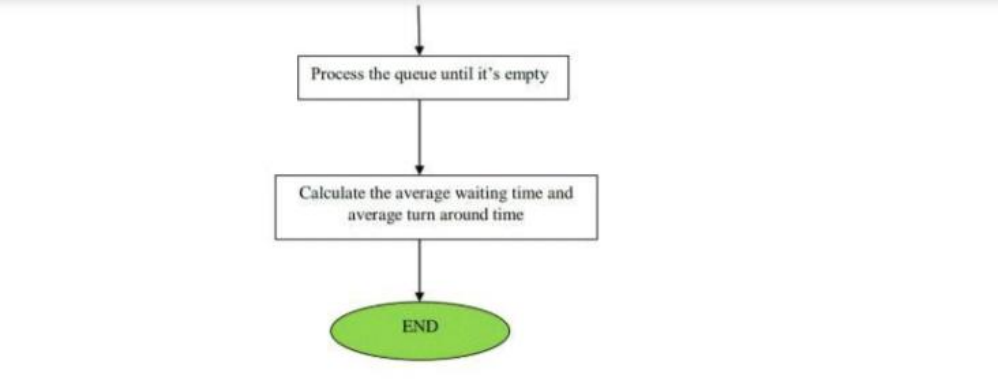
5. Our innovative algorithm(OIA)

- The proposed algorithm focuses on the improvement on CPU scheduling algorithm. The algorithm reduces the waiting time and turnaround time drastically compared to the other Scheduling algorithm and simple Round Robin scheduling algorithm.
- This proposed algorithm works in a similar way as but with some modification. It executes the shortest job having minimum burst time first instead of FCFS simple Round robin algorithm and it also uses Smart time quantum instead of static time quantum. Instead of giving static time quantum in the CPU scheduling algorithms, our algorithm calculates the Smart time quantum itself according to the burst time of all processes.
- The proposed algorithm eliminates the discrepancies of implementing simple round robin architecture.
- In the first stage of the innovative algorithm CPU scheduling algorithms all the processes are arranged in the increasing order of CPU burst time. It means it automatically assign the priority to the processes.

- Process having low burst time has high priority to the process have high burst time.
- Then in the second stage the algorithm calculates the mean of the CPU burst time of all the processes. After calculating the mean, it will set the time quantum dynamically i.e. (average of mean and highest burst time)/2.
- Then in the last stage algorithm pick the first process from the ready queue and allocate the CPU to the process for a time interval of up to 1 Smart time quantum. If the remaining burst time of the current running process is less than 1 Smart time quantum then algorithm again allocate the CPU to the Current process till it execution.
- After execution it will remove the terminated process from the ready queue and again go to the stage 3.

Flowchart for the proposed algorithm:





6. EXPERIMENT ANALYSIS:

➢ CPU Burst Time in Increasing Order

ALGORITHM	Average Waiting Time(ms)	Average Turnaround Time(ms)	Number Of Context Switch
FCFS	38.4	57.8	10
RR	30.4	49.8	7
SJF	26.4	45.8	6
Our innovative algorithm	24.4	43.7	4

0	0		2	10	16	24	0	0	6	14	0	0
P1		**P2**		**P3**	**P4**	**P5**	**P2**	**P3**	**P4**	**P5**	**P4**	**P5**
0	5		**15**	**25**	**35**	**45**	**47**	**57**	**67**	**77**	**83**	**97**

Gantt chart representation for FCFS

0	0		0	10	16	24	0		0	0
P1		**P2**		**P3**	**P4**	**P5**	**P3**		**P4**	**P5**
0	5		**17**	**27**	**37**	**47**	**57**		**73**	**97**

Gantt chart representation for RR

0	0		0	0	16	24	0	0
P1		**P2**		**P3**	**P4**	**P5**	**P4**	**P5**
0	5		**17**	**37**	**47**	**57**	**73**	**97**

Gantt chart representation for SJF

P1	P2	P3	P4	P5
0	5	17	37	63	97

Gantt chart representation for our innovative algorithm

➢CPU Burst Time in Decreasing Order

ALGORITHM	Average Waiting Time(ms)	Average Turnaround Time(ms)	Number Of Context Switch
FCFS	58	77.4	11
RR	49	68.4	8
SJF	34.4	53.8	7
Our innovative algorithm	24.4	43.79	4

0	24	16	10	2	0	14	6	0	0	4	0	0
P1	P2	P3	P4	P5	P1	P2	P3	P4	P1	P2	P1	
0	10	20	30	40	45	55	65	75	77	87	93	97

Gantt chart representation for FCFS

0	24	16	10	0	0	14	0	0	0
P1		P2	P3	P4	P5	P1	P2	P3	P1
0	10	20	30	42	47	57	73	83	97

Gantt chart presentation for RR

0	24	0	0	0	16	14	0	0
P1		P5	P4	P3	P2	P1	P2	P1
0	10	15	27	47	57	67	81	97

Gantt representation chart of SJF

0	0	0	0	0	8
P5		P4	P3	P2	P1
0	5	17	37	63	97

Gantt representation of our innovative algorithm

➢CPU Burst Time in Random Order

ALGORITHM	Average Waiting Time(ms)	Average Turnaround Time(ms)	Number Of Context Switch
FCFS	48	67.4	11
RR	40.4	59.8	8
SJF	31	50.4	6
Our innovative algorithm	24.4	43.7	4

0	10	24	0	2	16	0	14	0	6	4	0	0
P1		P2	P3	P4	P5	P1	P2	P3	P5	P2	P5	P2
0	10	20	25	35	45	55	65	67	77	87	93	97

Gantt chart representation for FCFS

0	10	24	0	0	16	0	14	0	0
P1		P2	P3	P4	P5	P1	P2	P5	P1
0	10	20	30	42	47	57	73	83	97

Gantt chart presentation for RR

0	0	0	0	16	24	0	0
P1		P3	P4	P5	P2	P5	P2
0	20	25	37	47	57	73	97

Gantt representation chart of SJF

0	0	0	0	0	8
P3		P4	P1	P5	P2
0	5	17	37	63	97

Gantt representation of Our innovative algorithm

7. CODE:

```
#include <stdio.h>
#include<math.h>
#include<string.h>
#include<limits.h>
int wt[100],bt[100],at[100],tat[100],n,p[100];
float awt[5],atat[5];
// we create this file to store all the processes information without slowing down the compiler
// by storing the data in a file and not the i/o buffer or temporary memory
void create_file_csv(char *filename,int a[][2],int n,int m)
{
printf("\n Creating %s.csv file",filename);
FILE *fp;
int i,j;
filename=strcat(filename,".csv");
fp=fopen(filename,"w+");
fprintf(fp,"FCFS, SJF, RR, SRTF, Innovative Algo");
```

```
for(i=0;i<m;i++)
{
fprintf(fp,"\n%d",i+1);
for(j=0;j<n;j++)
fprintf(fp,",%d ",a[i][j]);
}
fclose(fp);
printf("\n %sfile created",filename);
}
// used to enter the processes and their burst and arrival time
void input()
{
printf("Enter Number of processes:");
scanf("%d",&n);
int i;
for(i=0;i<n;i++)
p[i]=i+1;
for(i=0;i<n;i++)
{
printf("Enter Burst Time of process %d:",i+1);
scanf("%d",&bt[i]);
printf("Enter Arrival Time of process %d:",i+1);
scanf("%d",&at[i]);
}
for(i=0;i<5;i++)
{
awt[i]=0.0;
atat[i]=0.0;
}
}
//this is used to set the arrival time in increasing order
```

```
void changeArrival()
{
  int a=at[0];
  int i;
  for(i=0;i<n;i++)
  {
    if(at[i]<a)
    a=at[i];
  }
  if(a!=0)
  {
    for(i=0;i<n;i++)
    at[i]=at[i]-a;
  }
}
void fcfs()
{
//set the waiting time of first process to be 0
wt[0]=0;
//set the values of burst time ,turn around time and average turn aroundtime to be equal
atat[0]=tat[0]=bt[0];
int btt=bt[0];
int i;
for(i=0;i<n;i++){
//calculating the waiting time for the process
wt[i]=btt-at[i];
btt+=bt[i];
awt[0]+=wt[i]; //adding the value to average waiting time
tat[i]= wt[i]+bt[i]; // calculating the turnaround time
atat[0]+=tat[i]; //adding the value for the calculation of average turnaround time
}
```

```
//calculating the average calues for turnaround time and waiting time
atat[0]/=n;
awt[0]/=n;
printf("SR.\tA.T.\tB.T.\tW.T.\tT.A.T.\n");
for(i=0;i<n;i++)
{
printf("%3d\t%3d\t%3d\t%3d\t%4d\n",i+1,at[i],bt[i],wt[i],tat[i]);
}
}
void innovative(){
int i, total = 0, x, counter = 0, time_quantum;
int wait_time = 0, turnaround_time = 0, temp[100];
x=n;
for(i = 0; i < n; i++)
{
// used to check if bt of that process is equal to quantum
//if it is  then counter=1 indicating process is completed
temp[i] = bt[i];
}
// here we find ot the max bt
int avg,sum=0,max;
max=bt[0];
   for(int i=1; i<n; i++)
   {    sum=sum+bt[i];
        if(max<bt[i])
           max=bt[i];
   }
   avg=sum/n;
   //here we calculate the smart time quantum
   time_quantum=(max+avg)/2;
   printf(" smart quantum :%d\n",time_quantum);
```

```
printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t WaitingTime\n");
for(total = 0, i = 0; x != 0;)
{
if(temp[i] <= time_quantum && temp[i] > 0)
{
total = total + temp[i];
temp[i] = 0;
counter = 1;
}
// if the bt > than quantum then temp will hold the remaining bt after one quantum has been alotted to it
else if(temp[i] > 0)
{
temp[i] = temp[i] - time_quantum;
total = total + time_quantum;
}
// if a process is complete this condition is checked by its bt stored in temp =0 and the counter flag for keeping track of the completion is 1
//then we decrease x by 1 we use x as a flag in this loop to only exit when all the processes are finished
if(temp[i] == 0 && counter == 1)
{
x--;
printf("Process[%d]\t\t%d\t\t %d\t\t\t %d\n",i+1,bt[i],total-at[i],total-at[i]-bt[i]);
wait_time = wait_time + total - at[i] - bt[i];
turnaround_time = turnaround_time + total - at[i];
counter = 0;
}
// used to calculate the waiting time by the process by adding the total int to it we kept using in the prvious lines to know the time spent for the process
// and we calculate wt by sub at and bt from sum of wt and total
// here we set the counter back to zero for the next process we will evaluate
```

```
if(i == n - 1)
{//if we reach the end of the process array we switch back to the start of the process queue by setting i=0
i=0;
}
//we increment to the next process
else if(at[i+1]<=total)
{
i++;
}
else
{
i=0;
}
}
// here we calculate the avg wt and avg tat
awt[4]=wait_time*1.0/n;
atat[4]=turnaround_time*1.0/n;
}
void rr()
{
int i, total = 0, x, counter = 0, time_quantum;
int wait_time = 0, turnaround_time = 0, temp[100];
x=n;
for(i = 0; i < n; i++)
{
temp[i] = bt[i];
}
printf("\nEnter Time Quantum:\t");
scanf("%d", &time_quantum);
printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t WaitingTime\n");
```

```
for(total = 0, i = 0; x != 0;)
{
// used to check if bt of that process is equal to quantum
//if it is  then counter=1 indicating process is completed
if(temp[i] <= time_quantum && temp[i] > 0)
{
total = total + temp[i];
temp[i] = 0;
counter = 1;
}
// if the bt > than quantum then temp will hold the remaining bt after one quantum has been alotted to it
else if(temp[i] > 0)
{
temp[i] = temp[i] - time_quantum;
total = total + time_quantum;
}
// if a process is complete this condition is checked by its bt stored in temp =0 and the counter flag for keeping track of the completion is 1
//then we decrease x by 1 we use x as a flag in this loop to only exit when all the processes are finished
if(temp[i] == 0 && counter == 1)
{
x--;
printf("Process[%d]\t\t%d\t\t %d\t\t\t %d\n",i+1,bt[i],total-at[i],total-at[i]-bt[i]);
// used to calculate the waiting time by the process by adding the total int to it we kept using in the prvious lines to know the time spent for the process
// and we calculate wt by sub at and bt from sum of wt and total
wait_time = wait_time + total - at[i] - bt[i];
turnaround_time = turnaround_time + total - at[i];
// here we set the counter back to zero for the next process we will evaluate
counter = 0;
```

```
}
//if we reach the end of the process array we switch back to the start of the process queue by setting i=0
if(i == n - 1)
{
i=0;
}
//we increment to the next process
else if(at[i+1]<=total)
{
i++;
}
else
{
i=0;
}
}
// here we calculate the avg wt and avg tat
awt[2]=wait_time*1.0/n;
atat[2]=turnaround_time*1.0/n;
}
void findWaitingTime(int pid[],int bt[],int at[], int n, int wt[])
{ //this function is to calculate the waiting time for SRTF algorithm
        int rt[n];
        // we use rt array for storing remaining time
        for (int i = 0; i < n; i++)
                rt[i] = bt[i];
//complete is the flag we use for our while loop to keep it running till all processes are completed
        int complete = 0, t = 0, minm = INT_MAX;
        int shortest = 0, finish_time;
        int check =0;
```

```
        // Process until all processes gets
        // completed
        while (complete != n) {
                for (int j = 0; j < n; j++) {
        //we use this loop to find out which process had the shortest remaing time and store its index in shortest int
                        if ((at[j] <= t) && (rt[j] < minm) && rt[j] > 0) {
                                minm = rt[j];
                                shortest = j;
                                check =1;
                        }
                }
// if the rt for the process is not shortest then check will be 0 so we increase the t int for tracking the time spent with the shortest rt  and continue with the algorithm
                if (check == 0) {
                        t++;
                        continue;
                }
                //as we increment the t int we decrease the rt for shortest
                rt[shortest]--;
        //now the minm is equated to rt of shortest and checked if it is 0 meaning if the srt process is done if it is we increment the complete flag
                minm = rt[shortest];
                if (minm == 0)
                        minm = INT_MAX;
                        //as we increment the complete flag and we put check=0 and minm back to the value we initialized it to for the next process
                if (rt[shortest] == 0) {
                        complete++;
                        check =0;
                        // we calculate the finish time foe that process and equate it to t which we used to track time plus 1
```

```
                finish_time = t + 1;
                // we calculate the wt by sub bt and at from finsh time acting as the
total time consumed by the process
                wt[shortest] = finish_time - bt[shortest] - at[shortest];
// this is a situation to calc wt back to 0 if by an anamoly in values entered by the user the
wt becomes negative
                if (wt[shortest] < 0)
                        wt[shortest] = 0;
        }
        t++; }
}
void findTurnAroundTime(int bt[], int n, int wt[], int tat[])
{
    for (int i = 0; i < n; i++)
        tat[i] = bt[i] + wt[i];
}

void srtf(int pid[],int bt[],int at[], int n)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    findWaitingTime(p,bt,at, n, wt);
    findTurnAroundTime(bt, n, wt, tat);
    printf("Processes    Burst time    Waiting time    Turn around time\n");

    // Calculate total waiting time and
    // total turnaround time
    for (int i = 0; i < n; i++) {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];

    printf("  %d \t\t",pid[i]);
```

```
            printf("  %d \t\t",bt[i]);
            printf("  %d \t\n",tat[i]);
            }
double s=(float)total_wt / (float)n;
             //printf("Average waiting time = %f",s);
             awt[3]=s;
            double t=(float)total_tat / (float)n;
        //printf("Average turn around time = %f ",t);
        atat[3]=t;
}
void display(int c)
{

int i;
printf("Average Waiting Time: %f\nAverage Turn AroundTime:%f",awt[c-1],atat[c-1]);
}
void sjf()
{
float wavg=0,tavg=0,tsum=0,wsum=0;
int i,j,temp,sum=0,ta=0;
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if(at[i]<at[j])
{
//arranged the values according to the ascending order of arrival time
//swapped the values for the same
temp=p[j];
p[j]=p[i];
p[i]=temp;
```

```
temp=at[j];
at[j]=at[i];
at[i]=temp;
temp=bt[j];
bt[j]=bt[i];
bt[i]=temp;
}
}
}
int btime=0,min,k=1;
for(j=0;j<n;j++)
{
btime=btime+bt[j];
min=bt[k]; //set the value of min to the initial burst time
for(i=k;i<n;i++)
{
if(btime>=at[i] && bt[i]<min) //compared the processes with the min value set
{
//arranged the values according to the comparison result
temp=p[k];
p[k]=p[i];
p[i]=temp;
temp=at[k];
at[k]=at[i];
at[i]=temp;
temp=bt[k];
bt[k]=bt[i];
bt[i]=temp;
}
}
k++;  //increased the value of k
```

```
}
wt[0]=0;
for(i=1;i<n;i++)
{
sum=sum+bt[i-1];  //calculated the sum for the average burst time
wt[i]=sum-at[i];  // calculated the waiting time
wsum=wsum+wt[i]; //calculating the sum for average waiting time
}
awt[1]=(wsum/n);
for(i=0;i<n;i++)
{
ta=ta+bt[i];
tat[i]=ta-at[i];  //calculating the turnaround time
tsum=tsum+tat[i];  //calculating total turnaround time
}
atat[1]=(tsum/n);  //calculating average turnaround time
printf("SR.\tA.T.\tB.T.\tW.T.\tT.A.T.\n");
for(i=0;i<n;i++)
{
printf("%3d\t%3d\t%3d\t%3d\t%4d\n",i+1,at[i],bt[i],wt[i],tat[i]);
}
}
int main(){
printf("Welcome to CPU Scheduling:\n\n");
input();
int c,choice;
changeArrival();
printf("Choice\tAlgorithm used\n1\tFCFS Algorithm\n2\tSJF Algorithm\n3\tRound
robin\n4\tSRTF Algorithm\n5\tOur innovative algorithm\n");
do
{
```

```
printf("Enter your choice from the above table");
scanf("%d",&c);
switch(c)
{
case 1:fcfs();break;
case 2:sjf();break;
case 3:rr();break;
case 4:srtf(p,bt,at, n);break;
case 5:innovative();break;
default: printf("Please enter choice from 1 to 5 only\n");break;
}
display(c);
printf("\n\nEnter 1 to continue 0 to stop");
scanf("%d",&choice);
}while(choice==1);
int a[5][2],i;
for(i=0;i<5;i++)
{
a[i][0]=awt[i];
a[i][1]=atat[i];
}
create_file_csv("schedule",a,5,2);
}
```

8. OUTPUT

```
ubuntu@ubunu2004:~$ ./proj1
Welcome to CPU Scheduling:

Enter Number of processes:4
Enter Burst Time of process 1:2
Enter Arrival Time of process 1:4
Enter Burst Time of process 2:3
Enter Arrival Time of process 2:1
Enter Burst Time of process 3:3
Enter Arrival Time of process 3:5
Enter Burst Time of process 4:6
Enter Arrival Time of process 4:2
Choice  Algorithm used
1       FCFS Algorithm
2       SJF Algorithm
3       Round robin
4       SRTF Algorithm
5       Our innovative algorithm
```

- FCFS Output:

```
Enter your choice from the above table1
SR.     A.T.    B.T.    W.T.    T.A.T.
  1       3       2      -1       1
  2       0       3       4       7
  3       4       3       3       6
  4       1       6       9      15
Average Waiting Time: 3.750000
Average Turn AroundTime:7.750000

Enter 1 to continue 0 to stop1
```

- SJF Output:

```
Enter 1 to continue 0 to stop1
Enter your choice from the above table2
SR.     A.T.    B.T.    W.T.    T.A.T.
  1       0       3       0       3
  2       3       2       0       2
  3       4       3       1       4
  4       1       6       7      13
Average Waiting Time: 2.000000
Average Turn AroundTime:5.500000
```

- RR Output:

```
Enter 1 to continue 0 to stop1
Enter your choice from the above table3

Enter Time Quantum:     2

Process ID              Burst Time      Turnaround Time         WaitingTime
Process[1]              3               3                       0
Process[2]              2               2                       0
Process[3]              3               6                       3
Process[4]              6               13                      7
Average Waiting Time: 2.500000
Average Turn AroundTime:6.000000
```

- Our Innovative Algorithm:

```
Enter 1 to continue 0 to stop1
Enter your choice from the above table5
 smart quantum :4

Process ID              Burst Time      Turnaround Time         WaitingTime
Process[1]              3               3                       0
Process[2]              2               2                       0
Process[3]              3               4                       1
Process[4]              6               13                      7
Average Waiting Time: 2.000000
Average Turn AroundTime:5.500000
```

- SRTF:

```
Enter 1 to continue 0 to stop1
Enter your choice from the above table4
Processes       Burst time      Waiting time        Turn around time
  2              3              3
  1              2              2
  3              3              4
  4              6              13
Average Waiting Time: 2.000000
Average Turn AroundTime:5.500000

Enter 1 to continue 0 to stop0
```

9. CONCLUSION

- Results have shown that the proposed algorithm gives better results in terms of average waiting time, average turnaround time and number of context switches in all cases of process categories than the simple Round Robin CPU scheduling algorithm.

- In all these proposed algorithms time quantum is static due to which in these cases the number of context switches, average waiting time and average turnaround time will be very high and in our proposed algorithm, time quantum is calculated dynamically according to the burst time of all processes and it will find out a smart time quantum for all processes which gives good performance as compared to FCFS, RR and SJF.

10. REFERENCES:

1.Ankur B, Rachhpal S, Gaurav. Comparative Study of Scheduling Algorithms in Operating System. International Journal of Computers and Distributed Systems. Apr–May 2013; 3(1).

2.Raman, Pardeep Kumar Mittal. An Efficient Dynamic Round Robin CPU Scheduling Algorithm. IJARCSSE. May 2014; 4(5): 906–910p.

3. Silberschatz Abraham, Peter Galvin, Greg Gagne. Operating Systems Concepts with Java. 7th Edn. John Wiley & Sons; 2004.

4. Lingyun Yang, Schopf Jennifer M, Ian Foster T. Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments. Proceedings of the ACM/IEEE SC2003 Conference (SC) 1-58113-695-1/03 $ 17.00 © 2003 ACM.

5. Abbas Noon, Ali Kalakech, Seifedine Kadry. A New Round Robin Based Scheduling Algorithm for Operating Systems: Dynamic Quantum Using the Mean Average. IJCSI. May 2011; 8(3)(1): 224–229p.

6. Imran Qureshi. CPU Scheduling Algorithms: A Survey. Int J Advanced Networking and Applications (IJANA). 2014; 05(04): 1968–1973p.

7. Mostafa Samih M, Rida SZ, Hamad Safwat H. Finding Time Quantum of Round Robin CPU Scheduling Algorithm in General Computing Systems Using Integer Programming. International Journal of Research and Reviews in Applied Sciences (IJRRAS). 2010; 5(1): 64–71p.