

Scilab Manual for  
Operating Systems Lab  
by Dr Maheswari R  
Computer Engineering  
VIT CHENNAI<sup>1</sup>

Solutions provided by  
Dr Maheswari R  
Computer Engineering  
Vit Chennai

July 16, 2024

<sup>1</sup>Funded by a grant from the National Mission on Education through ICT, <http://spoken-tutorial.org/NMEICT-Intro>. This Scilab Manual and Scilab codes written in it can be downloaded from the "Migrated Labs" section at the website <http://scilab.in>



# Contents

List of Scilab Solutions	3
1 First Come First Serve Non pre-emptive CPU Scheduling using Scilab	6
2 Shortest Job First (SJF) Pre-emptive CPU Scheduling using Scilab	8
3 Graphical Analysis with Waiting time & Average waiting time of CPU Scheduling Algorithms using Scilab	10
4 Graphical Analysis with turn-around time & Average turnaround time of CPU Scheduling Algorithms using Scilab	14
5 Round Robin (RR) Pre-emptive CPU Scheduling using Scilab	18
6 Comparison of Various Partition Allocation Algorithms using Scilab	20
7 Deadlock Avoidance using Scilab	26
8 Process Synchronization Techniques using Scilab	28
9 Memory Management using Scilab	30
10 Page Replacement Algorithm using Scilab	35

# List of Experiments

Solution 1.0	First Come First Serve Non Preemptive CPU Scheduling using Scilab . . . . .	6
Solution 2.0	Shortest Job First . . . . .	8
Solution 3.0	Graphical Analysis WT and AWT . . . . .	10
Solution 4.0	Analysis TAT and ATAT . . . . .	14
Solution 5.0	Round Robin Scheduling . . . . .	18
Solution 6.0	Comparison of Various Partition Allocation Algorithms using Scilab . . . . .	20
Solution 7.0	Banker Algorithm Deadlock Avoidance using Scilab	26
Solution 8.0	Dekker Process Synchronization Techniques using Scilab . . . . .	28
Solution 9.0	Memory Management using Scilab First Fit Best Fit and Worst Fit . . . . .	30
Solution 10.0	Optimal Page Replacement Algorithm using Scilab	35
AP 1	Optimal page replacement . . . . .	40
AP 2	Worst Fit Memory Allocation . . . . .	41
AP 3	First Fit Memory Allocation . . . . .	42
AP 4	Display Function . . . . .	43
AP 5	Best Fit Memory Allocation . . . . .	44
AP 6	Dekker Algorithm . . . . .	47
AP 7	Deadlock Bankers Algorithm . . . . .	49
AP 8	Round Robin Scheduling New . . . . .	51
AP 9	Display Function of Round Robin . . . . .	51
AP 10	SJF algorithm for Turn Around Time and Average Turn Around Time calculation . . . . .	53
AP 11	Round Robin Scheduling for Turn Around Time and Average Turn Around Time calculation . . . . .	56

AP 12	FCFS Turn Around and Average Turn Around Calculation . . . . .	57
AP 13	SJF for Turn Around Time and Average Turn Around Time calculation . . . . .	58
AP 14	Round Robin Scheduling for Waiting and Average Waiting Time calculation . . . . .	61
AP 15	FCFS Waiting and Average Waiting Calculation .	62
AP 16	SJF New . . . . .	64
AP 17	Display Function SJF new . . . . .	66
AP 18	First Come First Serve CPU Scheduling . . . . .	67

# List of Figures

3.1	Graphical Analysis WT and AWT . . . . .	11
4.1	Analysis TAT and ATAT . . . . .	15
6.1	Comparison of Various Partition Allocation Algorithms using Scilab . . . . .	21

# Experiment: 1

## First Come First Serve Non pre-emptive CPU Scheduling using Scilab

check Appendix [AP 18](#) for dependency:

```
fcfs.sci
```

**Scilab code Solution 1.0** First Come First Serve Non Preemptive CPU Scheduling using Scilab

```
1 clear;
2 clc;
3 //WINDOWS 10 64-BIT OS , Scilab and toolbox versions
   6.1.0.
4
5 //Scheduling is a matter of managing queues and to
   decide which of the process have to be executed
   next to achieve high efficiency level.
6 //First Come First Serve (FCFS) Non Pre-emptive :
   Jobs are always executed on a first-come, first-
   serve basis.
```

```
7
8 //Functions to be loaded
9 exec("fcfs.sci");// fcfs.sci dependency file
10
11
12     num=4; //no of processes P1,P2,P3,P4
13
14     bt=[10 2 8 6]; //Sample burst time
15     wt=zeros(1,num); //waiting time
16     tat=zeros(1,num); //turn around time
17
18     disp("First Come First Serve (FCFS) Non Pre-
19         emptive CPU Scheduling");
20     disp("Burst time of the given Process P1=10, P2
21         =2, P3=8, P4=6");
22
23     disp('Waiting Time of each Process'); //displaying
24         the waiting time
25
26     fcfs = firstcomefirstserve(num,bt,wt,tat) //
27         Calling first come first serve function
```

---



## Experiment: 2

# Shortest Job First (SJF) Pre-emptive CPU Scheduling using Scilab

check Appendix [AP 17](#) for dependency:

```
display_sjf_new.sci
```

**Scilab code Solution 2.0** Shortest Job First

```
1 clear;  
2 clc;  
3  
4 //WINDOWS 10 64-BIT OS , Scilab and toolbox versions  
   6.1.0.  
5  
6 //SJF scheduling is employed when several processes  
   arrive almost at the same time, so as to avoid  
   conflict, ensure maximum CPU utilization with  
   minimum waiting time, turnaround time to minimize  
   starvation. The algorithm can be used for both
```

cases i.e., when arrival time is the same for all or most processes and when there are slightly different arrival times. In case of same arrival time, the values may be set to 0 by default by the user

```
7
8 // loading the necessary functions
9
10 exec("sjf_new.sci");
11 exec("display_sjf_new.sci");
12
13 num=4;           //no of processes P1,P2,P3,P4
14 pt=[10 2 8 6 ]; //process time or burst time
15 pid=[1 2 3 4]; //process id
16 wt=zeros(1,num); //waiting time
17 tat=zeros(1,num); //turn around time
18 total=0;        //total waiting time
19 total2=0;       //total turn around time
20
21 disp("Shortest Job First (SJF) Pre-emptive CPU
      Scheduling");
22 disp("Burst time of the given Process P1=10, P2=2,
      P3=8, P4=6");
23
24
25 disp("Sorted Process based on its Shortest Job");
26
27 sjf = shorestjobfirst(pid,num,pt,wt,tat); //
      Calling shorest job first function
```

---

check Appendix [AP 16](#) for dependency:

sjf\_new.sci

## Experiment: 3

# Graphical Analysis with Waiting time & Average waiting time of CPU Scheduling Algorithms using Scilab

check Appendix [AP 15](#) for dependency:

```
fcfs_wt_awt.sci
```

**Scilab code Solution 3.0** Graphical Analysis WT and AWT

```
1 clear;  
2 clc;  
3 //WINDOWS 10 64-BIT OS , Scilab and toolbox versions  
   6.1.0.  
4
```

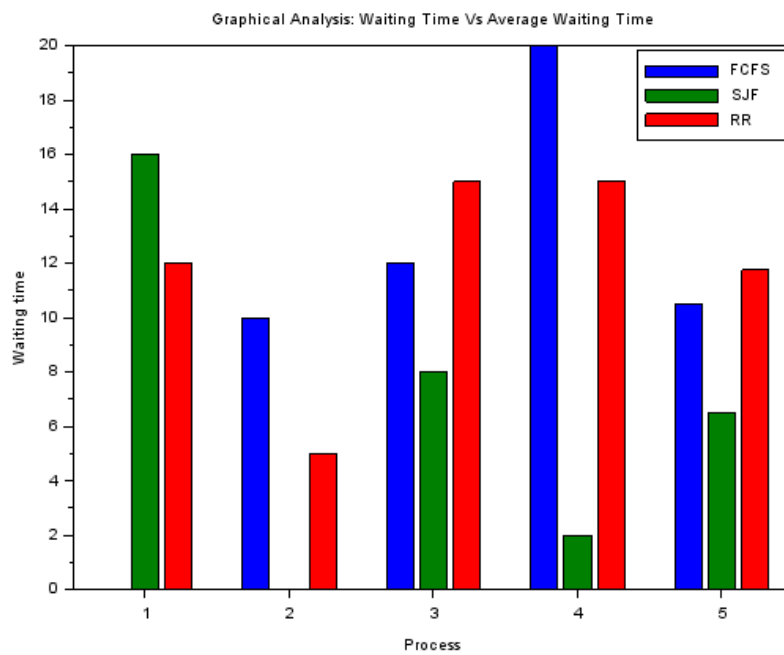


Figure 3.1: Graphical Analysis WT and AWT

```

5 //Scheduling algorithms deals to minimize queuing
  delay and to optimize performance of queuing
  environment. In this analysis , some common
  scheduling algorithms like First Come First Serve
  (FCFS), Shortest Job First (SJF) and Round Robin
  (RR) Scheduling are studied and reviewed on the
  basis of their working strategy
6
7
8 //Functions to be loaded
9 exec("fcfs_wt_awt.sci"); // fcfs_wt_awt.sci
  dependency file
10 exec("sjf_wt_awt.sci"); // sjf_wt_awt.sci dependency
  file
11 exec("rr_wt_awt.sci"); // rr_wt_awt.sci dependency
  file
12
13     num=4; //no of processes P1,P2,P3,P4
14     bt=[10 2 8 6]; //Sample burst time
15     wt=zeros(1,num); //waiting time
16     tat=zeros(1,num); //turn around time
17
18 disp("Graphical Analysis – Waiting Time vs Average
  Waiting Time of Scheduling Algorithms");
19
20 disp("Burst time of the given Process P1=10, P2=2,
  P3=8, P4=6");
21
22 disp('Waiting Time of each Process in FCFS'); //
  displaying the waiting time in FCFS
23
24 fcfs = firstcomefirstserve(num,bt,wt,tat) //
  Calling first come first serve function
25
26
27 disp('Waiting time of each Process in SJF'); //
  displaying the Waiting time of each Process in
  SJF

```

```

28 sjf = shortestjobfirst(num,bt,wt,tat)    // Calling
      shortest job first function
29
30
31 disp('Waiting Time of each Process in Round Robin');
      //displaying the Waititing Time of each Process
      in Round Robin
32
33 rr= roundrobin(num,bt,wt,tat)           // Calling
      Round Robin function
34
35 /*constructing a rows for graphical representation*/
36
37 scf(1);
38
39 y = [0,10,20,30,40];
40
41 x=[1,2,3,4,5]
42 avg =
      [0,16,12;10,0,5;12,8,15;20,2,15;10.5,6.5,11.75];
43
44 /*Matrix avg is set of values obtained from waiting
      time for each algorithm FCFS, SJF, RR
      respectively*/
45
46 xtitle('Graphical Analysis: Waiting Time Vs Average
      Waiting Time', 'Process', 'Waiting time');
47
48 bar(x, avg);
49
50 legend("FCFS", "SJF", "RR");

```

---

check Appendix [AP 14](#) for dependency:

rr\_wt\_awt.sci

check Appendix [AP 13](#) for dependency:

sjf\_wt\_awt.sci

## Experiment: 4

# Graphical Analysis with turn-around time & Average turnaround time of CPU Scheduling Algorithms using Scilab

check Appendix [AP 12](#) for dependency:

```
fcfs_tat_atat.sci
```

**Scilab code Solution 4.0** Analysis TAT and ATAT

```
1 clear;  
2 clc;  
3 //WINDOWS 10 64-BIT OS , Scilab and toolbox versions  
   6.1.0.  
4
```

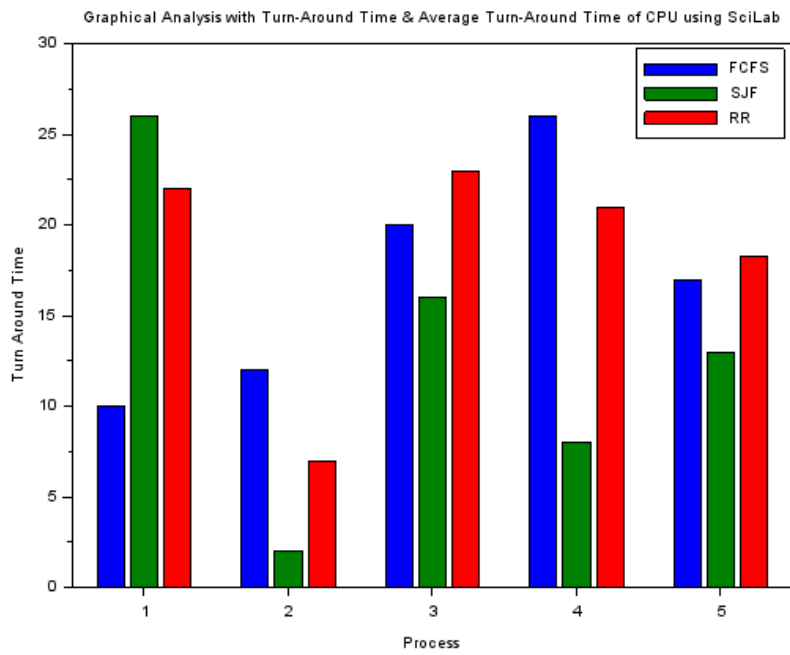


Figure 4.1: Analysis TAT and ATAT



```

5 //In this analysis , some common scheduling
  algorithms like First Come First Serve (FCFS),
  Shortest Job First (SJF) and Round Robin (RR)
  Scheduling are studied and reviewed on the basis
  of their Turn Around Time and Average Turn Around
  Time
6
7
8 //Functions to be loaded
9 exec("fcfs_tat_atat.sci"); // fcfs_tat_atat.sci
  dependency file for FCFS Scheduling
10 exec("sjf_tat_atat.sci"); // sjf_tat_atat.sci
  dependency file for SJF Scheduling
11 exec("rr_tat_atat.sci"); // rr_tat_atat.sci
  dependency file for RR Scheduling
12
13     num=4; //no of processes P1,P2,P3,P4
14     bt=[10 2 8 6]; //Sample burst time
15     wt=zeros(1,num); //waiting time
16     tat=zeros(1,num); //turn around time
17
18 disp("Graphical Analysis with Turn-Around Time &
  Average Turn-Around Time of CPU using SciLab");
19 disp("Burst time of the given Process P1=10, P2=2,
  P3=8, P4=6");
20
21 disp('Turn Around Time of each Process in FCFS'); //
  displaying the Turn Around time in FCFS
22
23 fcfs = firstcomefirstserve(num,bt,wt,tat) //
  Calling first come first serve function
24
25
26 disp('Turn Around Time of each Process in SJF'); //
  displaying the Turn Around time of each Process
  in SJF
27 sjf = shortestjobfirst(num,bt,wt,tat) // Calling
  shortest job first function

```

```

28
29
30 disp('Turn Around Time of each Process in Round
      Robin'); //displaying the Turn Around Time of
      each Process in Round Robin
31
32 rr= roundrobin(num,bt,wt,tat)           // Calling
      Round Robin function
33
34 /*constructing a rows for graphical representation*/
35
36 scf(1);
37
38 y = [0,10,20,30,40];
39
40 x=[1,2,3,4,5]
41 avg =
      [10,26,22;12,2,7;20,16,23;26,8,21;17,13,18.25];
42
43 /*Matrix avg is set of values obtained from Turn
      Around time for each algorithm FCFS, SJF, RR
      respectively*/
44
45 xtitle('Graphical Analysis with Turn-Around Time &
      Average Turn-Around Time of CPU using SciLab',
      'Process','Turn Around Time');
46
47
48 bar(x,avg);
49
50 legend("FCFS", "SJF", "RR");

```

---

check Appendix [AP 11](#) for dependency:

rr\_tat\_atat.sci

check Appendix [AP 10](#) for dependency:

sjf\_tat\_atat.sci

# Experiment: 5

## Round Robin (RR) Pre-emptive CPU Scheduling using Scilab

check Appendix [AP 9](#) for dependency:

```
display_rr.sci
```

**Scilab code Solution 5.0** Round Robin Scheduling

```
1 clear;  
2 clc;  
3  
4 //WINDOWS 10 64-BIT OS , Scilab and toolbox versions  
   6.1.0.  
5  
6 // Round Robin (RR) is a pre-emptive scheduling  
   algorithm. The CPU is shifted to the next process  
   after fixed interval time, which is called time  
   quantum/time slice.  
7
```

```

8 //Functions to be loaded
9 exec("roundrobin_new.sci");//dependency file
    roundrobin_new.sci
10 exec("display_rr.sci");//dependency file for display
    function
11
12 disp("                                ROUND ROBIN SCHEDULING
        ")
13
14 at = [0 1 2 3];    // Defining sample Arrival Time
15 bt = [9 5 3 4];    // Defining sample Burst Time
16 n=size(at);
17
18 disp(" Sample Quantum Time= 5 ")
19 mprintf("\n")
20 q = input("Enter Quantum Time: ");
21 disp(" Process                Turnaround time
        Waiting time");
22
23 //Calling Round Robin function
24 rr = roundrobin(q,n,at,bt);
25
26 //disp(" Process                Turnaround time
        Waiting time");

```

---

check Appendix [AP 8](#) for dependency:

roundrobin\_new.sci

# Experiment: 6

## Comparison of Various Partition Allocation Algorithms using Scilab

check Appendix [AP 5](#) for dependency:

```
best_fit_func.sci
```

check Appendix [AP 4](#) for dependency:

```
display_func.sci
```

check Appendix [AP 3](#) for dependency:

```
first_fit_func.sci
```

**Scilab code Solution 6.0** Comparison of Various Partition Allocation Algorithms using Scilab

```
1 clear;
```

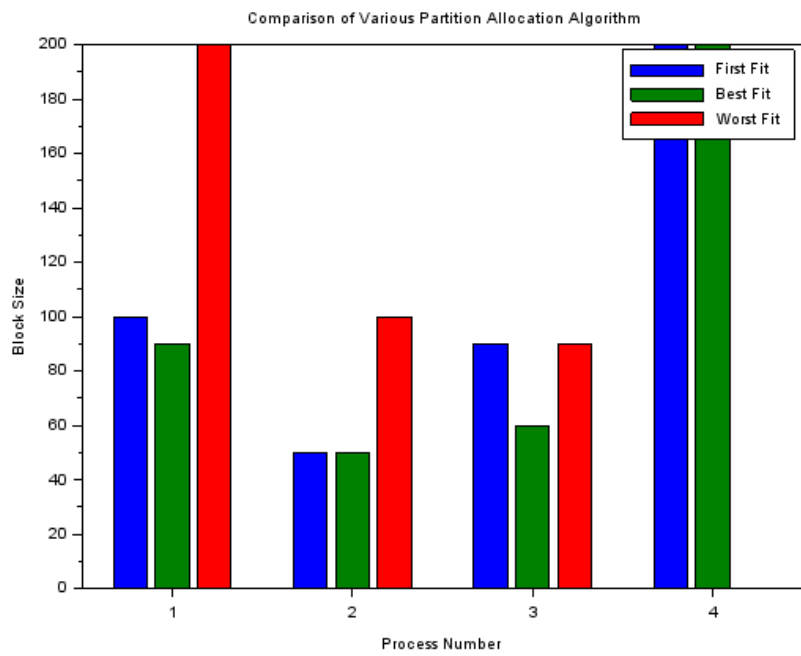


Figure 6.1: Comparison of Various Partition Allocation Algorithms using Scilab

```

2  clc;
3  //WINDOWS 10 64-BIT OS , Scilab and toolbox versions
    6.1.0.
4
5  //This experiment is compare the various partition
    allocation algorithms used by the operating
    system for memory allocation
6  //    1. First-Fit Memory Allocation  2. Best-Fit
    Memory Allocation 3. Worst-Fit Memory Allocation
7
8  // loading all the necessary functions
9  exec("first_fit_func.sci");
10 exec("best_fit_func.sci");
11 exec("worst_fit_func.sci");
12 exec("display_func.sci");
13
14 // Example problem
15 p = [90 20 50 200]; // Defining sample Process Size
16 b = [50 100 90 200 60]; // Defining sample Block Size
17
18
19 //Determining the number of processes and blocks
20
21 size_process = size(p); // Size of the process
    array is calculated using size() function
22 size_process = size_process(2);
23
24 size_block = size(b); // Size of the block
    array is calculated using size() function
25 size_block = size_block(2);
26
27 /*calling the function, defined in first fit.sci,
    for first fit allocation*/
28 ff_allot = firstFit(p,b,size_process,size_block)
29
30 /*calling the function, defined in best fit.sci, for
    best fit allocation*/
31 bf_allot = bestFit(p, b, size_process, size_block)

```

```

32
33 /*calling the function, defined in worst fit.sci,
    for worst fit allocation*/
34 wf_allot = worstFit(p,b,size_process,size_block);
35
36
37 ff_allotsize = zeros(1,size_process); //
    ff_allotsize - size of the selected blocks for
    first fit
38 bf_allotsize = zeros(1,size_process); //
    bf_allotsize - size of the selected blocks for
    best fit
39 wf_allotsize = zeros(1,size_process); //
    wf_allotsize - size of the selected blocks for
    worst fit
40
41 // storing the allocated block size for each process
    according to the respective fits
42
43 for i=1:size_process
44     if ff_allot(i)~=0 then //
        checking if any block is selected
45         ff_allotsize(i) = b(ff_allot(i)); //
            storing the size of the selected block
            for first fit
46     else
47         ff_allotsize(i) = 0 //
            store size as 0 if no block is selected
48     end,
49
50     if bf_allot(i)~=0 then //
        checking if any block is selected
51         bf_allotsize(i) = b(bf_allot(i)); //
            storing the size of the selected block
            for best fit
52     else
53         bf_allotsize(i) = 0 //
            store size as 0 if no block is selected

```



```

54     end,
55
56     if wf_allot(i)~=0 then //
        checking if any block is selected
57         wf_allotsize(i) = b(wf_allot(i)); //
            storing the size of the selected block
            for worst fit
58     else
59         wf_allotsize(i) = 0; //
            store size as 0 if no block is selected
60     end,
61 end
62
63
64 //constructing a matrix(y) for graphical
    representation of first fit , best fit and worst
    fit
65
66 y= zeros(size_process,3);
67
68 y(:,1) = ff_allotsize; // First fit allotment
    size
69 y(:,2) = bf_allotsize; // Best fit allotment
    size
70 y(:,3) = wf_allotsize; // Worst fit allotment
    size
71
72 bar(y) // plotting a bar graph
73
74 xtitle("Comparison of Various Partition Allocation
    Algorithm", "Process Number", "Block Size");
75 legend("First Fit" , "Best Fit", "Worst Fit" );
76
77
78 //printing first fit , best fit and worst fit array
79
80 mprintf("Comparison of Various Partition Allocation
    Algorithms using Scilab\n\n") //displaying the

```

```

    title of experiment
81 mprintf("Ex: Process size P1=90, P2=20, P3=50, P4
    =200"); //displaying
    the sample Process size considered for the of
    experiment
82 mprintf("\n    Block or hole size B1= 50, B2= 100,
    B3=90, B4=200\n"); //displaying
    the sample Block size considered for the of
    experiment
83
84 mprintf("\nFIRST FIT:\n")
85 mprintf("Process no. \tProcess size\tBlock no. Block
    size\n")
86 display(ff_allot,ff_allotsize,size_process,p)
    //displaying the process and block
    allocation by first fit array
87
88 mprintf("\nBEST FIT:\n")
89 mprintf("Process no. \tProcess size\tBlock no. Block
    size\n")
90 display(bf_allot,bf_allotsize,size_process,p)
    //displaying the process and block
    allocation by best fit array
91
92
93 mprintf("\nWORST FIT:\n")
94 mprintf("Process no. \tProcess size\tBlock no. Block
    size\n")
95 display(wf_allot,wf_allotsize,size_process,p)
    //displaying the process and block
    allocation by worst fit array

```

---

check Appendix [AP 2](#) for dependency:

worst\_fit\_func.sci

# Experiment: 7

## Deadlock Avoidance using Scilab

check Appendix [AP 7](#) for dependency:

```
Deadlock.sci
```

**Scilab code Solution 7.0** Banker Algorithm Deadlock Avoidance using Scilab

```
1 clear;
2 clc;
3
4 //The bankers algorithm is a resource allocation
   and deadlock avoidance algorithm that tests for
   safety by simulating the allocation for
   predetermined maximum possible amounts of all
   resources
5
6 // Load dependency file
7 exec("Deadlock.sci");
8
```

```

9 //WINDOWS 10 64-BIT OS , Scilab and toolbox versions
   6.1.0.
10 close;
11 n=5; // Number of processes
12 m=3; // Number of resources
13
14
15 disp(" Bankers Algorithm for Deadlock Avoidance")
16 mprintf("\nConsider Number of Processes N=5,Number
   of resources M=3\n"); // Number of processes and
   Number of resources
17
18 mprintf("\n Allocation are [0 1 0;2 0 0;3 0 2;2 1
   2;0 0 2]"); // Giving the process allocation
   values
19 mprintf("\nM=[7 5 3;3 2 2;9 0 2;2 2 2;4 3 3]\n"); //
   Giving the maximum values
20
21 disp('Following is the SAFE Sequence satisfies the
   safety requirement:'); // Disp command prints the
   safe sequence
22
23 Deadlock(n,m) // Function call of Deadlock –
   Bankers Algorithm

```

---

# Experiment: 8

## Process Synchronization Techniques using Scilab

check Appendix [AP 6](#) for dependency:

Process\_Synch.sci

**Scilab code Solution 8.0** Dekker Process Synchronization Techniques using Scilab

```
1  ///DEKKER'S ALGORITHM///
2
3  clear;
4  clc;
5  //WINDOWS 10 64-BIT OS , Scilab and toolbox versions
   6.1.0.
6
7  exec("Process_Synch.sci");
8
9
10 //Dekker's algorithm guarantees mutual exclusion ,
    freedom from deadlock , and freedom from
    starvation .
```

```

11 //In this algorithm , the position of each process is
    indicated with the variables turn and flag.
12
13
14 //INITIALIZING THE VALUES
15 c1=1,c2=1,turn=1;
16 count0=0,count1=0,count=0;
17 i=0;
18 //Deafult values
19 limit=10;    //Limit of CS
20 a=1,b=1;    //Time taken in CS by both process
21
22
23 disp("Process Synchronization using Dekkers
    Algorithm in Scilab");
24 mprintf("\nInput Ex: Time in CS for P1 is 2, P2 is
    3, Total Time is 9\n ")
25 //INSTRUCTIONS TO PROCEED :
26 mprintf(" 1) Process 1 enters \n");
27 mprintf(" 2) Process 2 enters \n");
28 mprintf(" 3) Both process enters \n");
29 mprintf(" 4) Exit \n");
30 mprintf("\n Enter total time requied by Process 1 in
    CS:")
31 a = input(" ")
32 mprintf(" Enter total time requied by Process 2 in
    CS:")
33 b = input(" ")
34 mprintf(" Enter total time limit of CS :")
35 limit = input(" ")
36 //Psyn(a,b,limit);
37 Psyn(a,b,limit)
38 //FUNCTION TO IDENTIFY WHICH PROCESS ENTERS NOW

```

---

# Experiment: 9

## Memory Management using Scilab

check Appendix [AP 5](#) for dependency:

```
best_fit_func.sci
```

check Appendix [AP 4](#) for dependency:

```
display_func.sci
```

check Appendix [AP 3](#) for dependency:

```
first_fit_func.sci
```

**Scilab code Solution 9.0** Memory Management using Scilab First Fit Best Fit and Worst Fit

```
1 clear;
```

```

2  clc;
3  //WINDOWS 10 64-BIT OS , Scilab and toolbox versions
   6.1.0.
4  //The memory to the processor will be allocated in
   several blocks of memory, in order to make a
   perfectly organized allocation between the memory
   blocks and process , three different partition
   and allocation algorithms are used 1. First-Fit
   Memory Allocation 2. Best-Fit Memory Allocation
   3. Worst-Fit Memory Allocation
5
6  /* loading all the necessary functions */
7  exec("first_fit_func.sci");
8  exec("best_fit_func.sci");
9  exec("worst_fit_func.sci");
10 exec("display_func.sci");
11
12 mprintf("Memory Management First fit , Best Fit and
   Worst Fit Allocation\n ");
13 mprintf("
   _____\
   n ");
14
15 mprintf("Ex: Process size P1=212, P2=417, P3=112, P4
   =426");
16 mprintf(" \n      Block or hole size B1= 100, B2= 500,
   B3=200, B4=300, B5=600\n");
17
18 /* Example problem */
19 p = [212,417,112,426]; // process size
20 b = [100,500,200,300,600]; // block or hole size
21
22 disp("Select the Option:");
23 mprintf(" 1-First Fit\n");
24 mprintf(" 2-Best Fit \n");
25 mprintf(" 3-Worst Fit \n");
26
27 /*

```



```

28 Determining the number of processes and blocks
29 */
30 size_process = size(p);
31 size_process = size_process(2);
32
33 size_block = size(b);
34 size_block = size_block(2);
35
36 n1=input('');
37 if(n1==1) then
38 // firstfit();
39 /*calling the function, defined in first fit.sci,
    for first fit allocation*/
40 ff_allot = firstFit(p,b,size_process,size_block)
41 end
42 if(n1==2) then
43 /*calling the function, defined in best fit.sci, for
    best fit allocation*/
44 bf_allot = bestFit(p, b, size_process,
    size_block)
45
46 end
47 if(n1==3) then
48 // worstfit();
49 /*calling the function, defined in worst fit.sci,
    for worst fit allocation*/
50 wf_allot = worstFit(p,b,size_process,size_block);
51
52 end
53
54 /*
55 ff_allotsize - size of the selected blocks for first
    fit
56 bf_allotsize - size of the selected blocks for best
    fit
57 wf_allotsize - size of the selected blocks for worst
    fit
58 */

```

```

59 ff_allotsize = zeros(1,size_process);
60 bf_allotsize = zeros(1,size_process);
61 wf_allotsize = zeros(1,size_process);
62
63 /* storing the allocated block size for each process
    according to the respective fits */
64 for i=1:size_process
65     if(n1==1) then
66
67         if ff_allot(i)~=0 then // checking if any
            block is selected
68             ff_allotsize(i) = b(ff_allot(i)); //
                storing the size of the selected
                block for first fit
69         else
70             ff_allotsize(i) = 0 // store size as 0
                if no block is selected
71         end,
72     end
73
74     if(n1==2) then
75         if bf_allot(i)~=0 then // checking if any
            block is selected
76             bf_allotsize(i) = b(bf_allot(i)); //
                storing the size of the selected
                block for best fit
77         else
78             bf_allotsize(i) = 0 // store size as 0
                if no block is selected
79         end,
80     end
81     if(n1==3) then
82         if wf_allot(i)~=0 then // checking if any
            block is selected
83             wf_allotsize(i) = b(wf_allot(i)); //
                storing the size of the selected
                block for worst fit
84         else

```

```

85             wf_allotsize(i) = 0; // store size as 0
                if no block is selected
86         end,
87     end
88 end
89 /*
90 printing first fit, best fit and worst fit array
91 */
92 if(n1==1) then
93     mprintf("\nFIRST FIT:\n")
94     mprintf("Process no. \tProcess size\tBlock no. Block
        size\n")
95     display(ff_allot,ff_allotsize,size_process,p)
96 end
97 if(n1==2) then
98
99     mprintf("\nBEST FIT:\n")
100    mprintf("Process no. \tProcess size\tBlock no.
        Block size\n")
101    display(bf_allot,bf_allotsize,size_process,p)
102 end
103 if(n1==3) then
104
105    mprintf("\nWORST FIT:\n")
106    mprintf("Process no. \tProcess size\tBlock no.
        Block size\n")
107    display(wf_allot,wf_allotsize,size_process,p)
108 end

```

---

check Appendix [AP 2](#) for dependency:

worst\_fit\_func.sci

# Experiment: 10

## Page Replacement Algorithm using Scilab

check Appendix [AP 1](#) for dependency:

PageReplacement.sci

**Scilab code Solution 10.0** Optimal Page Replacement Algorithm using Scilab

```
1 clear;
2 clc;
3 //WINDOWS 10 64-BIT OS , Scilab and toolbox versions
   6.1.0.
4
5 // In Optimal page replacement algorithm , the page
   that will not be used for the longest period of
   time is replaced to make space for the requested
   page.
6
7 //Functions to be loaded
```

```

8  exec("PageReplacement.sci"); // PageReplacement.sci
   dependency file
9
10 mprintf(" Optimal Page Replacement Algorithm using
   Scilab\n")
11
12 mprintf(" Sample Input: No. of frames=4, No. Page=13
   ")
13 mprintf("\n")
14 mprintf("Sample Page Reference String
   7,0,1,2,0,3,0,4,2,3,0,3,2")
15 frames=cell(10); //Read Frames
16 pages=cell(30); //Read Pages
17 temp=cell(10); //Read temp
18 noOfFrames=0;
19 noOfPages=0;
20
21 mprintf("\n")
22 mprintf('Enter No Of Frames'); //Write to command
   window
23 mprintf("\n")
24 noOfFrames=input(""); // Giving the no.of.Frames
   Value
25
26 mprintf('Enter No Of Pages'); //Write to command
   window
27 noOfPages=input(""); // Giving the no.of.Pages
   Value
28 flag1=0; //Setting a Flag
29 flag2=0; //Setting a Flag
30 flag3=0; //Setting a Flag
31 faults=0; //Read faults
32 maximum=0; //Setting it to a marker value
33
34 mprintf("\n")
35 mprintf('Enter Page Reference Values'); //Write to
   command window
36

```

```
37 for n=1:noOfPages //Giving the first reference value
    in first frame
38 // mprintf(" Enter Page reference %d",n)
39 pages{n}=input(""); //Printing the value in
    page
40 end
41
42 pagereplacement(noOfFrames ,noOfPages ,pages) //
    Calling pagereplacement function
```

---

# Appendix

## Scilab code AP 1

```
1 function []= pagereplacement(noOfFrames,noOfPages
    ,pages)  ////Page replacement function
2
3 for n=1:noOfFrames //Giving the no. of Frames
4     frames{n}=-1;
5 end
6
7 for i=1:noOfPages // No. of Pages
8     flag1=0;
9     flag2=0;
10        for j=1:noOfFrames
11            if(frames{j}==pages{i})
12                flag1=1;
13                flag2=1;
14                break;
15            end
16        end
17        if(flag1==0)
18            for j=1:noOfFrames
19                if(frames{j}==-1)
20                    faults=faults+1;
21                    frames{j}=pages{i};
22                    flag2=1;
23                    break;
24                end
25            end
26        end
27    end
28 end
```

```

26     end
27     if(flag2==0)
28         flag3=0;
29         for j=1:noOfFrames
30             temp{j}=-1;
31             for k=i+1:noOfPages
32                 if(frames{j}==pages{k}) //Check the
                                     frames and pages are filled
33                     temp{j}=k
34                     break;
35                 end
36             end
37         end
38
39         //For loop for identifying page faults
40         for j=1:noOfFrames
41             if(temp{j}==-1)
42                 pos=j;
43                 flag3=1;
44                 break;
45             end
46         end
47         if(flag3==0)
48             maximum=temp{0};
49             pos=0;
50             for j=1:noOfFrames
51                 if(temp{j}>maximum)
52                     maximum=temp{j}; //Check the
                                     value is long period use or
                                     not
53                     pos=j;
54                 end
55             end
56         end
57         frames{pos}=pages{i}
58         faults=faults+1; //Check next value is
                             Equal or to change the position
59     end

```



```

60 end
61 faults=faults-1;
62 mprintf("No Of Page Faluts %d\n",faults); //Write
    Page Faults to command window
63
64 endfunction

```

Optimal page replacement

---

### Scilab code AP 2

```

1 //Worst-Fit Memory Allocation, the operating
    system searches the entire list and allocates the
    largest available hole to the process.
2 //If a large process comes at a later stage, then
    memory may not have space to accommodate it.
3
4 function [wf_allot]=worstFit(p,b,size_process,
    size_block)
5
6 // Declaring worst fit flag array(wf_flag) which is
    used for maintaining the status of each block(
    free or busy)
7
8     wf_allot = zeros(1,size_process);
    // Declaring worst fit array(wf_allot)
9     wf_flag = zeros(1,size_block);
10
11 //For loop for allocating blocks according to worst
    fit
12
13     for i=1:size_process
14         k = -1; // k - index
            position of the largest block which can
            accommodate a process, initially set to
            -1
15         for j=1:size_block
16             if p(i)<=b(j) && wf_flag(j) == 0 then //
                if process size is less than block

```

```

size and block is free
17     if k==-1 then
18         k = j;           // update k with
                           index position of the block
19     elseif(b(k)<b(j)) // if there is a
                           larger block which can
                           accommodate the process
20         k = j; // update k with the
                           index position of the larger
                           block
21     end,
22     end,
23 end
24 if(k==-1)
25     wf_allot(i)=0; // if no block can
                           accomodate the process, set allotted
                           block number as 0
26 else
27     wf_allot(i) = k; // store the selected
                           index in the worst fit array in the
                           index position i(process number)
28     wf_flag(k) = 1; // set the status of the
                           selected block as busy
29     end,
30 end
31 endfunction

```

#### Worst Fit Memory Allocation

---

**Scilab code AP 13** //First-Fit Memory Allocation  
algorithm scans the memory and whenever it finds  
the first big enough hole to store a process ,  
2 //it stops scanning and loads the process into that  
hole/block .  
3  
4 **function** [ff\_allot]=firstFit(p,b,size\_process ,  
size\_block)  
5

```

6
7 //Declaring first fit flag array( ff_flag) which is
   used for maintaining the status of each block(
   free or busy)
8
9   ff_allot = zeros(1,size_process);           //
   Declaring first fit array(ff_allot)
10  ff_flag = zeros(1,size_block);
11
12  // For loop for allocating blocks according to
   first fit
13
14  for i=1:size_process
15    for j=1:size_block
16      if p(i) <= b(j) && ff_flag(j)==0 then
   // if process size is less than block
   size and block is free
17        ff_allot(i) = j;
   // store index position of the block
   in ff_allot in the index position i(
   process number)
18        ff_flag(j) = 1;
   // set status as busy
19        break
20      end,
21    end
22  end
23 endfunction

```

First Fit Memory Allocation

---

#### Scilab code AP 4

```

1 //Display Function:It prints all required details
   such as Process no. , Process size , Block no. ,
2 //Block size for First Fit, Best Fit and Worst Fit.
3
4 function display(allot,allotsize,size_process,p)
5   for i=1:size_process

```

```

6     if allot(i)==0 then
7         mprintf("P%d\t\t%d\t\t Not allocated
                -\n",i,p(i))           // Display
                the Process number that could not
                allocated
8     else
9         mprintf("P%d\t\t%d\t\tB%d\t\t %d\n",i,p(i)
                ),allot(i),allotsize(i)) // Display
                the Process number with allocated
                Block number
10        end,
11    end
12 endfunction

```

Display Function

---

```

Scilab code AP 15 //Best Fit Memory Allocation , the
                operating system searches the whole memory
                according to the size of the given process
2 //and allocates it to the smallest hole which is big
                enough to accommodate it.
3
4
5 function [bf_allot]=bestFit(p,b,size_process ,
                size_block)
6
7
8 //    declaring best fit flag array(bf_flag) which
                is used for maintaining the status of each block(
                free or busy)
9
10    bf_allot = zeros(1,size_process);           //
                declaring best fit array(bf_allot)
11    bf_flag = zeros(1,size_block);
12
13    // For loop for allocating blocks according to
                best fit
14    for i=1:size_process

```

```

15         k = -1;                                // k -
           index position of the smallest block
           which can accommodate a process ,
           initially set to -1
16     for j=1:size_block
17         if p(i)<=b(j) && bf_flag(j) == 0 then //
           if process size is less than block
           size and block is free
18             if k==-1 then
19                 k = j;                          // update k
           with index position of the
           block
20             elseif (b(j)<b(k))                  // if there
           is a smaller block which can
           accommodate the process
21                 k = j;                          // update k
           with the index position of
           the smaller block
22             end,
23         end,
24     end
25     if(k==-1)
26         bf_allot(i)=0;                          // if no
           block can accommodate the process , set
           allotted block number as 0
27     else
28         bf_allot(i) = k;                        // store the
           selected index in the best fit array
           in the index position i(process
           number)
29         bf_flag(k) = 1;                        // set the
           status of the selected block as busy
30     end,
31 end
32 endfunction

```

Best Fit Memory Allocation

---

### Scilab code AP 6

```
1 //In this program Dekkers Algorithm is used ,
  to ensure one process enters the critical section
  at a time while the other processes need to wait
  for the first one to leave the critical section.
2
3 function [] = Psyn(a,b,limit)
4 while (i<=limit)
5     printf("\n")
6     x = input("SELECT THE OPTION : ")
7
8     //PROCESS 1 ENTERS
9     if x==1 then // Option 1
10        c1=0;
11        while c2==0
12            if turn==2 then
13                c1=1;
14                while turn==2 //do nothing
15                    end
16                c1=0;
17            end
18        end
19        //critical section
20        count0=count0+a;
21        i=i+a;
22        // yield
23        c1=1;
24        turn=2;
25        //remainder section
26        if (count0>limit) | (i>limit) then
27            printf("Exceeds the limit of CS\n")
28            printf("END\n");
29            i = 100;
30        else
31            if(a>0)
32                printf("Process P1 Enters the
33                    Critical section");
34            printf("\nTotal Time of P1 in
```

```

34         Critical Section :%d\n",count0);
35     end
36     printf("\nIt is the turn process P2\n
37         ");
38     end
39 //PROCESS 2 ENTERS
40 if x==2 then // Option 2
41     c2=0;
42     while c1==0
43         if turn==1
44             c2=1;
45             while turn==1 //do nothing
46                 end
47             c2=0;
48         end
49     end
50 //critical section
51 count1=count1+b;
52 i=i+b;
53 // yield
54 c2=1;
55 turn=1;
56 //remainder section
57 if (count1>limit) | (i>limit) then
58     printf("Exceeds the limit of CS\n")
59     printf("END\n");
60     i = 100;
61 else
62     if(b>0)
63         printf("Process P2 Enters the
64             Critical section");
65         printf("\nTotal Time of P2 in
66             Critical Section :%d\n",count1);
67     end
68     printf("\nIt is the turn process P1\n
69         ");

```

```

67         end
68     end
69
70     //BOTH PROCESS ENTER AT SAME TIME Option 3
71     if x==3 then
72         printf("\nBoth process cant enter at same
73             time in Critical Section\n");
74         if i>limit then
75             printf("END\n");
76             i = 100;
77         end
78     end
79     //END for OTHER CONDITIONS Option 4
80     if x==4 then
81         printf("\nEND\n");
82         i = 100;
83     end
84 end
85 endfunction

```

Dekker Algorithm

---

Scilab code AP 17 // Deadlock – Bankers Algorithm

```

2
3 function [] = Deadlock(n,m)
4
5 // P0 P1 P2 P3 and P4 are the process names.
6 A=[0 1 0;2 0 0;3 0 2;2 1 2;0 0 2]; // Giving the
7   process allocation values
8 M=[7 5 3;3 2 2;9 0 2;2 2 2;4 3 3]; // Giving the
9   maximum values
10 L=[3 3 2]; // This gives the available values
11 ind=0;
12 ans1=list();
13 z=1;
14 f=list();
15

```



```

14 for k=1:5 // Iterating the values for all the 5
    processes
15     f(k)=0;
16 end
17
18 for i=1:5 //Iterating the values for all the
    processes
19     for j=1:3 //Iterating the value for all the
        processes
20         need(i,j)= M(i,j) - A(i,j); // Need is
            calculated by subtracting the maximum and
            available resources
21
22     end;
23
24 end
25 y=0;
26 for k=1:5 // Iterating for all the 5 processes
27     for i=1:5
28         if(f(i)==0) then
29             flag = 0; //Flag value is set zero
30             for j=1:3 // Iterating for all the resources
31                 if (need(i,j) > L(j)) then // If the
                    need value is more than the Available
                    resources , the request cannot be
                    granted
32                     flag=1; // Then the flag is set to 1
33                     break; // The loop breaks here
34                 end
35             end
36             if (flag==0) then // If the condition is
                satisfied the next process is checked
                similarly
37                 ans1(z)=i;
38                 z=z+1;
39
40             for y=1:3
41                 L(y)= L(y)+A(i,y); //If the condition is

```

```

        satisfied the available valueis
        updated by adding available value and
        the allocated resources of the
        particular process
42         end
43         f(i)=1;
44     end
45 end
46 end
47 end
48
49 //For loop for displaying SAFE Sequence which
    satisfies the safety
50 for i=1:5
51     ans1(i)=ans1(i)-1;
52 end
53 for i=1:5
54
55     mprintf('    <P%d>,' ,ans1(i));
56
57 end;
58 endfunction

```

Deadlock Bankers Algorithm

---

```

Scilab code AP 8 //Function for RoundRobin Algorithm
2
3 function [tat,wait_time]=roundrobin(q,n,at,bt)
4
5 remain = n//Storing no of process in a variable
    called n
6
7 wait_time=0;
8 tat=0;
9
10 quantum_time=q;
11 //disp(" Process           Turnaround time
    Waiting time");

```

```

12
13 time=0; //completion time is initially set to zero
14
15 for i=1:4
16     rt(i)=bt(i);
17 end;
18
19 //running the processes for specified quantum
20 while remain~=0
21     for i=1:4
22
23         if rt(i)<=quantum_time & rt(i)>0 then //
                executes if burst time is greater than 0
                and lesser than quantum time
24             time=time+rt(i); // update completion
                time
25             rt(i)=0;
26             flag=1;
27         elseif rt(i)>0 then
28             rt(i)=rt(i)-quantum_time; //update
                burst time
29             time=time+quantum_time;
30         end;
31         if rt(i)==0 & flag==1 then //executes if
                burst time is equal to 0 and flag=1
32             remain = remain-1;
33             mprintf('\n P%i\t\t\t %i\t\t\t %i',i
                ,time-at(i),time-at(i)-bt(i));
34             tat=tat+time-at(i); //Turnaround time
                = completion time-arrival time
35             wait_time = wait_time+time-at(i)-bt(
                i); //Waiting time = turnaround-
                burst time
36             flag=0;
37         end;
38         if i==n-1
39             i=1;
40         elseif at(i)<=time then //executes when

```

```

        arrival time is lesser than/equal to
        completion time
41         i=1;
42         else
43         i=1;
44         end;
45     end;
46 end;
47
48 // display function call
49 d = displayfunc(tat,wait_time); //Average
    Displaying turn around time and Average waiting
    time
50
51 endfunction

```

Round Robin Scheduling New

---

#### Scilab code AP 9

```

1 function [atat,awt] = displayfunc(tat,wait_time)
2     awt=wait_time*1.0/4; //Total wait_time/no of
    processes gives average waiting time,
    similarly avg turnaround time is calculated
3     atat=tat*1.0/4;
4
5     mprintf("\n")
6
7     disp(" Average Waiting Time using RR= "); //
    Displaying Average waiting time
8     disp(awt);
9
10    disp(" Average Turnaround Time using RR= "); //
    Displaying Average Turnaround time
11    disp(atat);
12 endfunction

```

Display Function of Round Robin

---

```

Scilab code AP 10 // SJF Scheduling function
2
3 function [tat,wait_time]=shortestjobfirst(num,btime,
    wtime,tatime)
4     total=0;           //total waiting time
5     total2=0;
6     n=num;
7     ptime=btime;
8     process=[1 2 3 4]; //process id
9     fd = %io(2);
10
11 for i=1:1:n-1 //sorting the processes in terms of
    process times
12     for j=i+1:1:n
13         if(ptime(i)>ptime(j))
14             temp=ptime(i);
15             ptime(i) = ptime(j);
16             ptime(j) = temp;
17             temp = process(i);
18             process(i) = process(j);
19             process(j) = temp;
20         end
21     end
22
23 end
24
25 wtime(1) = 0;
26 for i=2:1:n
27     wtime(i) = wtime(i-1)+ptime(i-1); //wait time
        of a process is sum of wait time of process
        before it and process time of process before
        it
28     total = total + wtime(i);           //finding
        total waiting time
29 end
30
31 tatime(1) = 0;
32 for i=1:1:n

```

```

33     tatime(i)=ptime(i)+wtime(i);    //turn around
        time=burst time +wait time
34     total2=total2+tatime(i);        //total
        turn around time
35 end
36
37 avg1 = total2/n;                    //finding
        average time
38
39 for i=1:1:n
40     fprintf(fd, ' P%d is %d',process(i),tatime(i))
        ;
41 end
42
43 fprintf(fd, '\n Average Turn-Around Time in SJF
        %.2f ', avg1);
44
45 endfunction

```

SJF algorithm for Turn Around Time and Average Turn Around Time calculation

---

```

Scilab code AP III // Round Robin Scheduling function
2
3 function [tat,wait_time]=roundrobin(num,btime,wtime,
    tatime)
4 b=0;
5 t=0;
6 n=num
7 q=5; //quantum time
8 wtime=zeros(1,n); //waiting time
9 fd = %io(2);
10 rtime=btime // burst time
11
12 //For loop : running the processes for specified
    quantum
13
14 for i=1:1:n //running the processes for 1

```

```

quantum
15     if(runtime(i)>=q)
16         for j=1:1:n
17             if(j==i)
18                 runtime(i)=runtime(i)-q;    //setting
                                                the remaining time if it is the
                                                process scheduled
19             else if(runtime(j)>0)
20                 wtime(j)=wtime(j)+q;    //
                                                incrementing wait time if it
                                                is not the process scheduled
21             end
22         end
23     end
24 else if(runtime(i)>0)
25     for j=1:1:n
26         if(j==i)
27             runtime(i)=0;                //as the
                                                remaining time is less than
                                                quantum it will run the process
                                                and end it
28         else if(runtime(j)>0)
29             wtime(j)=wtime(j)+runtime(i);
                                                //incrementing wait time if
                                                it is not the process
                                                scheduled
30         end
31     end
32 end
33 end
34 end
35 end
36 for i=1:1:n
37     if(runtime(i)>0)                //if remaining time is left
38         set flag
39         flag=1;
40     end
end

```

```

41 while(flag==1)           //if flag is set run the
    above process again
42     flag=0;
43     for i=1:1:n
44         if(rtime(i)>=q)
45             for j=1:1:n
46                 if(j==i)
47                     rtime(i)=rtime(i)-q;
48                 else if(rtime(j)>0)
49                     wtime(j)=wtime(j)+q;
50                 end
51             end
52         end
53     else if(rtime(i)>0)
54         for j=1:1:n
55             if(j==i)
56                 rtime(i)=0;
57             else if(rtime(j)>0)
58                 wtime(j)=wtime(j)+rtime(
                    i);
59             end
60         end
61     end
62 end
63 end
64 end
65 for i=1:1:n
66     if(rtime(i)>0)
67         flag=1;
68     end
69 end
70 end
71
72 //For loop : calculating turn around time for each
    process
73 for i=1:1:n
74     tatime(i)=wtime(i)+btime(i);    //By adding
        waiting time and burst time

```



```

75 end
76 for i=1:1:n
77     b=b+wtime(i);
78     t=t+tatime(i);
79 end
80
81 for i=1:1:n
82     mfprintf(fd, ' P%d is %d',i,tatime(i));
83 end
84
85 //displaying the Average Turn-Around Time in RR
86
87 mfprintf(fd, '\n Average Turn-Around Time in RR
88           %.2f ',t/n);
89 endfunction

```

Round Robin Scheduling for Turn Around Time and Average Turn Around Time calculation

---

Scilab code AP 12

```

function [tat,wait_time]=
    firstcomefirstserve(num,btime,wtime,tatime) //
    Function defintion of first come first serve
2
3 t1=0; // intializing time t1=0 for
    total waiting time calculation
4 t2=0; // intializing time t2=0 for
    total turn round time calculation
5
6 btime = bt; // assigning burst time
7 wtime = wt; // assigning waiting time
8 tatime = tat // assigning turn around time
9 n=num; // assigning number of process n
    =4 here
10 fd = %io(2);
11
12 //For loop for calculating total turn around time
    of each Process

```

```

13  for i=2:1:n
14      wtime(i)=btime(i-1)+wtime(i-1); //waiting time
           will be sum of burst time of previous process
           and waiting time of previous process
15      t1=t1+wtime(i); //calculating
           total time
16  end
17
18  for i=1:1:n
19      tatime(i)=btime(i)+wtime(i); //turn around
           time=burst time +wait time
20      t2=t2+tatime(i); //total turn
           around time
21  end
22
23      for i=1:1:n
24          mfprintf(fd, ' P%d is %d',i,tatime(i));
25      end
26
27
28  mfprintf(fd, '\n Average Turn-Around Time in FCFS
           %.2 f ',t2/n);
29
30  endfunction

```

FCFS Turn Around and Average Turn Around Calculation

---

Scilab code AP 13 // SJF Scheduling

```

2
3  function [tat,wait_time]=shortestjobfirst(num,btime,
           wtime,tatime)
4      total=0; //total waiting time
5      n=num;
6      ptime=btime;
7      process=[1 2 3 4]; //process id
8      fd = %io(2);
9

```

```

10 for i=1:1:n-1 //sorting the processes in terms of
    process times
11     for j=i+1:1:n
12         if(ptime(i)>ptime(j))
13             temp=ptime(i); // assigning Temporary
                variable for sorting
14             ptime(i) = ptime(j);
15             ptime(j) = temp;
16             temp = process(i);
17             process(i) = process(j);
18             process(j) = temp;
19         end
20     end
21
22 end
23
24 wtime(1) = 0;
25 for i=2:1:n
26     wtime(i) = wtime(i-1)+ptime(i-1); //wait time
        of a process is sum of wait time of process
        before it and process time of process before
        it
27     total = total + wtime(i); //finding
        total waiting time
28 end
29
30 avg = total/n; //finding
    average time
31
32 for i=1:1:n
33     fprintf(fd, ' P%d is %d', process(i), wtime(i));
34 end
35
36 fprintf(fd, '\n Average Waiting Time in SJF is
    %.2 f ', avg);
37
38 endfunction

```

## SJF for Turn Around Time and Average Turn Around Time calculation

---

### Scilab code AP 14 // Round Robin Scheduling

```
2 function [tat,wait_time]=roundrobin(num,btime,wtime,
   tatype)
3 b=0;
4 t=0;
5 n=num
6 q=5; //quantum time
7 wtime=zeros(1,n); //waiting time
8 fd = %io(2);
9 rtime=btime
10
11 //running the processes for specified quantum
12 for i=1:1:n //running the processes for
   quantum =5
13     if(rtime(i)>=q)
14         for j=1:1:n
15             if(j==i)
16                 rtime(i)=rtime(i)-q; //setting
   the remaining time if it is the
   process scheduled
17             else if(rtime(j)>0)
18                 wtime(j)=wtime(j)+q; //
   incrementing wait time if it
   is not the process scheduled
19             end
20         end
21     end
22     else if(rtime(i)>0)
23         for j=1:1:n
24             if(j==i)
25                 rtime(i)=0; //as the
   remaining time is less than
   quantum it will run the process
   and end it
26             else if(rtime(j)>0)
```

```

27             wtime(j)=wtime(j)+rtime(i);
                //incrementing wait time if
                it is not the process
                scheduled
28             end
29         end
30     end
31 end
32 end
33 end
34 for i=1:1:n
35     if(rtime(i)>0)           //if remaining time is left
        set flag
36         flag=1;
37     end
38 end
39
40 //Total waiting time , processes average waiting
    time calculation
41 while(flag==1)           //if flag is set run the
    above process again
42     flag=0;
43     for i=1:1:n
44         if(rtime(i)>=q)
45             for j=1:1:n
46                 if(j==i)
47                     rtime(i)=rtime(i)-q;
48                 else if(rtime(j)>0)
49                     wtime(j)=wtime(j)+q;
50                 end
51             end
52         end
53     else if(rtime(i)>0)
54         for j=1:1:n
55             if(j==i)
56                 rtime(i)=0;
57             else if(rtime(j)>0)
58                 wtime(j)=wtime(j)+rtime(

```

```

59                                     i); // Updation of
60                                     waiting time for each
61                                     process
62                                     end
63                                     end
64                                     end
65                                     end
66                                     end
67                                     end
68                                     end
69                                     end
70                                     end
71
72 //displaying the waiting time in RR
73
74 for i=1:1:n
75     mfprintf(fd, ' P%d is %d',i,wtime(i));
76 end
77
78 // Calculating Average Waiting Time
79 for i=1:1:n
80     b=b+wtime(i);
81     t=t+tatime(i);
82
83 end
84
85 //displaying the Average waiting time in RR
86
87 mfprintf(fd, '\n Average Waiting Time in Round
88 Robin is %.2f ',b/n);
89 endfunction

```

Round Robin Scheduling for Waiting and Average Waiting Time calculation

```

Scilab code AP 15 function [tat,wait_time]=
    firstcomefirstserve(num,btime,wtime,tatime) //
    Function defintion of first come first serve
2
3 t1=0; // intializing time t1=0 for
    total waiting time calculation
4 t2=0; // intializing time t2=0 for
    total turn round time calculation
5
6 btime = bt; // assigning burst time
7 wtime = wt; // assigning waiting time
8 tatime = tat // assigning turn around time
9 n=num; // assigning number of process n
    =4 here
10 fd = %io(2);
11
12 //For loop for calculating total waiting time of
    each Process
13 for i=2:1:n
14     wtime(i)=btime(i-1)+wtime(i-1); //waiting time
        will be sum of burst time of previous
        process and waiting time of previous process
15     t1=t1+wtime(i); //calculating
        total waiting time
16 end
17
18
19 //displaying the waiting time of each Process
20 for i=1:1:n
21     mfprintf(fd,' P%d is %d',i,wtime(i));
22 end
23
24 mfprintf(fd,'\n Average Waiting Time in FCFS is
        %.2 f ',t1/n);
25
26 endfunction

```

FCFS Waiting and Average Waiting Calculation

---

Scilab code AP 16 //WINDOWS 10 64-BIT OS , Scilab and  
toolbox versions 6.1.0.

```
2
3 // loading the necessary functions
4 function [tat,wait_time]=shorestjobfirst(pid,num,pt,
    wt,tat) // Function defintion of first come
    first serve
5
6 process=pid; //process id
7 n=num; // number of processes
8 ptime = pt; //process time or burst time
9 tatime =tat; //turn around time
10 wtime = wt; //waiting time
11 fd = %io(2);
12
13 //Determining the number of processes and blocks
14 size_process = size(process);
15 size_process = size_process(2);
16 size_ptime = size(ptime);
17 size_ptime = size_ptime(2);
18
19 //marks the position of process with minimum burst
    time in the specified range. This may be used to
    rearrange the order of the processes to achieve
    proper SJF scheduling...
20 for i=1:1:n-1 //For loop for sorting the processes
    in terms of process times
21     for j=i+1:1:n
22         if(ptime(i)>ptime(j))
23             temp=ptime(i); //temporary
                variable used to enable efficient
                swapping of values ..
24             ptime(i) = ptime(j);
25             ptime(j) = temp;
26             temp = process(i);
27             process(i) = process(j);
28             process(j) = temp;
29         end
```



```

30     end
31
32 end
33
34     wtime(1) = 0;
35     //waiting time calculation
36     for i=2:1:n
37         wtime(i) = wtime(i-1)+ptime(i-1);    //wait time
           of a process is sum of wait time of process
           before it and process time of process before
           it
38         total = total + wtime(i);           //finding
           total waiting time
39     end
40
41     //total turnaround time calculation
42     for i=1:1:n
43         tatetime(i)=ptime(i)+wtime(i);    //turn around
           time=burst time +wait time
44         total2=total2+tatetime(i);        //total
           turn around time
45     end
46
47     avg = total/n;                         //finding
           average time, average waiting time calculated by
           dividing total waiting time by number of proceses
48     avg1 = total2/n;                       //average
           turn around time calculated by dividing total
           turn around time by number of processes
49
50     display(process ,size_process ,wtime ,tatetime ,avg ,avg1);
           //displaying the process and block
           allocation by first fit array
51 endfunction

```

SJF New

---

```

Scilab code AP 117 //WINDOWS 10 64-BIT OS , Scilab and
    toolbox versions 6.1.0.
2 //Display Function:It prints all required details
    such as Process no.
3 //Waiting time, Turn-Around time, Average Waiting
    time and Average Turn-Around time
4
5 function display(process,size_process,wtime,tatime,
    avg,avg1)
6         //display of final values
7
8 for i=1:1:n
9     mfprintf(fd,' P%d',process(i)); //
        Displaying sorted Process based on its
        Shortest Job
10 end
11
12 disp('Waiting time of each Process using SFJ'); //
    displaying the Waiting time
13 for i=1:1:n
14     mfprintf(fd,' P%d is %d',process(i),wtime(i));
15 end
16
17 disp('Turn-Around time of each Process using SFJ');
    //displaying the Turn-Around time
18 for i=1:1:n
19     mfprintf(fd,' P%d is %d',process(i),tatime(i))
        ;
20 end
21
22 mfprintf(fd,'\n Average Waiting Time using SJF is
    %.2f',avg); //displaying the Average Waiting
    time
23 mfprintf(fd,'\n Average Turn-Around Time using SFJ
    is %.2f',avg1); //displaying the Average Turn-
    Around time
24
25 endfunction

```

```

Scilab code AP 18 function [tat,wait_time]=
    firstcomefirstserve(num,btime,wtime,tatime) //
    Function defintion of first come first serve
2
3 t1=0; // intializing time t1=0 for
    total waiting time calculation
4 t2=0; // intializing time t2=0 for
    total turn round time calculation
5
6 btime = bt; // assigning burst time
7 wtime = wt; // assigning waiting time
8 tatime = tat // assigning turn around time
9 n=num; // assigning number of process n
    =4 here
10 fd = %io(2);
11
12 //For loop for calculating total waiting time of
    each Process
13 for i=2:1:n
14     wtime(i)=btime(i-1)+wtime(i-1); //waiting time
        will be sum of burst time of previous
        process and waiting time of previous process
15     t1=t1+wtime(i); //calculating
        total waiting time
16 end
17
18 //For loop for calculating total turn around time of
    each Process
19 for i=1:1:n
20     tatime(i)=btime(i)+wtime(i); //turn around
        time=burst time +wait time
21     t2=t2+tatime(i); //total turn
        around time
22 end
23

```

```

24
25 //displaying the waiting time of each Process
26 for i=1:1:n
27     mfprintf(fd,' P%d is %d',i,wtime(i));
28 end
29
30 disp('Turn-Around Time of each Process');
    //displaying the final Turn-Around time of
    each Process
31 for i=1:1:n
32     mfprintf(fd,' P%d is %d',i,tatime(i));
33 end
34
35 mfprintf(fd,'\n Average Waiting Time is %.2f'
    ,t1/n); //displaying the Average
    waiting time
36 mfprintf(fd,'\n Average Turn-Around Time is %
    .2f',t2/n); //displaying the Average Turn
    Around time
37
38 endfunction

```

First Come First Serve CPU Scheduling

---