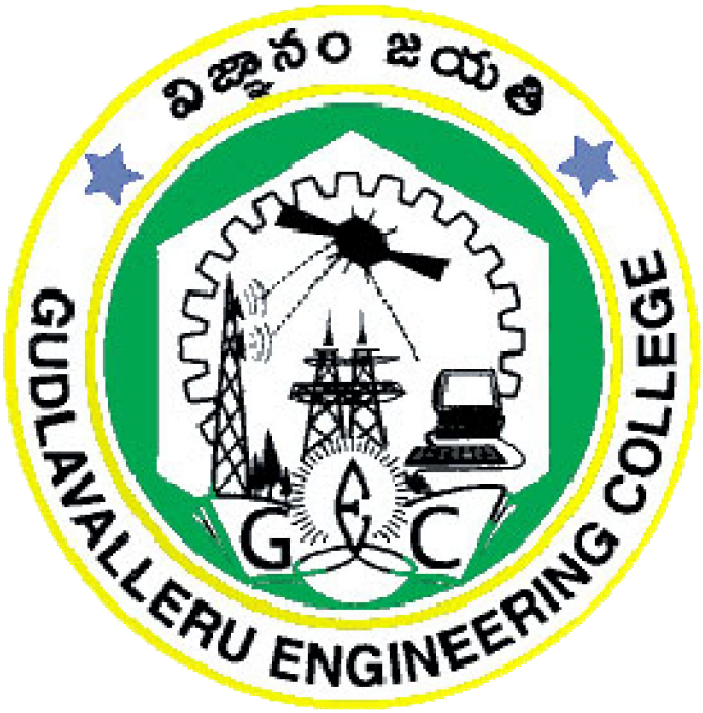
OPERATING SYSTEMS LAB

FACULTY MANUAL

II Year II Semester



Prepared by

Manasa. Y

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GUDLAVALLERU ENGINEERING COLLEGE

(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)

Seshadri rao Knowledge Village, Gudlavalleru – 521356

GUDLAVALLERU ENGINEERING COLLEGE

(An Autonomous Institution with Permanent Affiliation to JNTUK, Kakinada)

Seshadri Rao Knowledge Village, Gudlavalleru – 521356

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE VISION & MISSION

INSTITUTE VISION:

To be a leading institution of engineering education and research, preparing students for leadership in their fields in a caring and challenging learning environment.

INSTITUTE MISSION:

- To produce quality engineers by providing state-of-the-art engineering education.
- To attract and retain knowledgeable, creative, motivated and highly skilled individuals whose leadership and contributions uphold the college tenets of education, creativity, research and responsible public service.
- To develop faculty and resources to impart and disseminate knowledge and information to students and also to society that will enhance educational level, which in turn, will contribute to social and economic betterment of society.
- To provide an environment that values and encourages knowledge acquisition and academic freedom, making this a preferred institution for knowledge seekers.
- To provide quality assurance.
- To partner and collaborate with industry, government, and R&D institutes to develop new knowledge and sustainable technologies and serve as an engine for facilitating the nation's economic development.
- To impart personality development skills to students that will help them to succeed and lead.
- To instil in students the attitude, values and vision that will prepare them to lead lives of personal integrity and civic responsibility.
- To promote a campus environment that welcomes and makes students of all races, cultures and civilizations feel at home.
- Putting students face to face with industrial, governmental and societal challenges.

DEPARTMENT VISION & MISSION

VISION

To be a Centre of Excellence in Computer Science and Engineering education and training to meet the challenging needs of the industry and society.

MISSION

- To impart quality education through well-designed curriculum in tune with the growing software needs of the industry.
- To serve our students by inculcating in them problem solving, leadership, teamwork skills and the value of commitment to quality, ethical behavior & respect for others.
- To foster industry-academia relationship for mutual benefit and growth.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs):-

PEO1: Identify, analyze, formulate and solve Computer Science and Engineering problems both independently and in a team environment by using the appropriate modern tools.

PEO2: Manage software projects with significant technical, legal, ethical, social, environmental and economic considerations

PEO3: Demonstrate commitment and progress in lifelong learning, professional development, leadership and communicate effectively with professional clients and the public.

PROGRAM OUTCOMES (POs)

Engineering students will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a

member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES

Students will be able to

PSO1: Design, develop, test and maintain reliable software systems and intelligent systems.

PSO2: Design and develop web sites, web apps and mobile apps.

Course Objectives:

- To develop the concepts of process and memory management techniques.
- To know the problems of deadlock and study the various handling mechanisms.

Course Outcomes:

Upon successful completion of the course, the students will be able to

- implement CPU and disk scheduling algorithms.
- develop code for memory management techniques.
- develop code to implement Bankers algorithm to avoid deadlocks.

Mapping Of Course Outcomes With Program Outcomes

OPERATING SYSTEMS LAB	1	2	3	4	5	6	7	8	9	10	11	12	PSO1	PSO2
CO1: implement CPU and disk scheduling algorithms	3	2	2					2	2	2			2	
CO2: develop code for memory management techniques	2	2	2					2	2	2			2	
CO3: develop code to implement Bankers algorithm to avoid deadlocks	2	2	2					2	2	2			2	
Operating Systems lab	3	3	3					3	3	3			3	

LIST OF EXPERIMENTS

ADDITIONAL LAB EXPERIMENTS

EXERCISE: 1

AIM: 1(a) Write a program to implement the First come First Serve (FCFS) CPU scheduling algorithm.

DESCRIPTION:

- First Come First Serve (FCFS) is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival.
- It is the easiest and simplest CPU scheduling algorithm.

ALGORITHM:

Step1: Input the processes along with their burst time (bt).

Step 2: Find waiting time (wt) for all processes.

Step 3: As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.

Step 4: Find waiting time for all other processes i.e. for process i -> wt[i] = bt[i-1] + wt[i-1] .

Step 5: Find turnaround time = waiting_time + burst_time for all processes.

Step 6: Find average waiting time = total_waiting_time / no_of_processes.

Step 7: Similarly, find average turnaround time = total_turn_around_time / no_of_processes

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
        int n,bt[20],i,wt[20],tt[20],stt=0,swt=0;
        float awt,att;

        printf("Enter number of process : ");
        scanf("%d",&n);
        printf("Enter process cpu time : ");
        for(i=0;i<n;i++)
        scanf("%d",&bt[i]);
        wt[0]=0;
        tt[0]=stt=bt[0];
        for(i=1;i<n;i++)
```

```
        {
                wt[i]=bt[i-1]+wt[i-1];
                tt[i]=wt[i]+bt[i];
                swt+=wt[i];
                stt+=tt[i];
        }

        printf("Cpu time\tWaiting time\tTurn around time");
        for(i=0;i<n;i++)
        {
                printf("\n%d\t\t%d\t\t%d",bt[i],wt[i],tt[i]);
        }
        awt=(float)swt/n;
        att=(float)stt/n;
        printf("\nAverage waiting time : %f",awt);
        printf("\nAverage turn around time : %f\n",att);


}
```

OUTPUT:

```
Enter number of process : 5
Enter process cpu time : 2 3 6 5 4
Cpu time        Waiting time            Turn around time
2                       0                       2
3                       2                       5
6                       5                       11
5                       11                      16
4                       16                      20
Average waiting time : 6.800000
Average turn around time : 10.800000
```

VIVA QUESTIONS:

1. What are the shortcomings with FCFS CPU scheduling algorithm?
2. What is Convoy Effect?

AIM: 1(b) Write a program to implement the Shortest Job First (SJF) CPU scheduling algorithm.

DESCRIPTION:

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution.

- Two schemes:

- **Non preemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst
- **Preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)

PROGRAM:

```
 #include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
        int n,bt[20],i,j,t,wt[20],tt[20],swt=0,stt=0;
        float awt,att;
        printf("Enter number of process : ");
        scanf("%d",&n);
        printf("Enter process cpu time : ");
        for(i=0;i<n;i++)
        scanf("%d",&bt[i]);
        printf("Cpu time before sorting : ");
        for(i=0;i<n;i++)
        printf("%d ",bt[i]);
        printf("\nCpu time after sorting : ");
        for(i=0;i<n;i++)
        {
                        for(j=i+1;j<n;j++)
                        {
                                if(bt[i]>bt[j])
                                {
                                        t=bt[i];
                                        bt[i]=bt[j];
                                        bt[j]=t;
                                }
                        }
        }
        for(i=0;i<n;i++)
```

```
                printf("%d ",bt[i]);

                wt[0]=0;
                tt[0]=stt=bt[0];
                for(i=1;i<n;i++)
                {
                        wt[i]=bt[i-1]+wt[i-1];
                        tt[i]=wt[i]+bt[i];
                        swt+=wt[i];
                        stt+=tt[i];
                }
        printf("\nCpu time\tWaiting time\tTurn around time");
        for(i=0;i<n;i++)
        printf("\n%d\t\t%d\t\t%d",bt[i],wt[i],tt[i]);

        awt=(float)swt/n;
        att=(float)stt/n;
        printf("\nAverage waiting time : %f\n",awt);
        printf("Average turn around time : %f\n",att);

}
```

OUTPUT:

```
Enter number of process : 4
Enter process cpu time : 3 2 4 1
Cpu time before sorting : 3 2 4 1
Cpu time after sorting : 1 2 3 4
Cpu time    Waiting time            Turn around time
1                   0                       1
2                   1                       3
3                   3                       6
4                   6                       10
Average waiting time : 2.500000
Average turn around time : 5.000000
```

VIVA QUESTIONS:

1. What is the advantage of SJF algorithm?
2. What is the drawback of SJF algorithm?
3. What is the difference between preemptive SJF and non-preemptive SJF?
4. What is the other name for non-preemptive SJF?
5. What is the other name for preemptive SJF?

AIM: 1(c) Write a program to implement the Priority based process scheduling algorithm.

DESCRIPTION:

- A priority is associated with each process, and the CPU is allocated to the process with highest priority.
- Equal-priority processes are scheduled in FCFS order.
- The large CPU burst, the lower the priority, and vice versa.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7, or 0 to 4095. Here we assume low numbers represent high priority.

ALGORITHM:

Algorithm for Priority Based:

1- First input the processes with their burst time and priority.

2- Sort the processes, burst time and priority according to the priority.

3- Now simply apply FCFS algorithm.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{

        int n,bt[10],wt[10],pt[10],tt[10],swt=0,stt=0,t,t1,i,j;
        float awt,att;
        printf("Enter number of process : ");
        scanf("%d",&n);
        printf("Enter %d process times : ",n);
        for(i=0;i<n;i++)
        scanf("%d",&bt[i]);
        printf("Enter %d process priorites : ",n);
        for(i=0;i<n;i++)
        scanf("%d",&pt[i]);
        printf("Cpu time and priorities before sorting\n");
        for(i=0;i<n;i++)
        printf("%d\t%d\n",bt[i],pt[i]);
        printf("Cpu times and priorities after sorting\n");
        for(i=0;i<n;i++)
        {
                for(j=i+1;j<n;j++)
                {
                        if(pt[i]>pt[j])
                        {
```

```
                        t=pt[i];
                        pt[i]=pt[j];
                        pt[j]=t;
                        t1=bt[i];
                        bt[i]=bt[j];
                        bt[j]=t1;
                    }
                }
            }
            for(i=0;i<n;i++)
                printf("%d\t%d\n",bt[i],pt[i]);

                wt[0]=0;
                tt[0]=stt=bt[0];
                for(i=1;i<n;i++)
                {
                    wt[i]=bt[i-1]+wt[i-1];
                    tt[i]=wt[i]+bt[i];
                    swt+=wt[i];
                    stt+=tt[i];
                }
                awt=(float)swt/n;
                att=(float)stt/n;
                printf("Cpu time\tPriority\tWaiting time\tTurn around time\n");
                for(i=0;i<n;i++)
                printf("%d\t\t%d\t\t%d\t\t%d\n",bt[i],pt[i],wt[i],tt[i]);
                printf("Average waiting time : %f\n",awt);
                printf("Average turn around time : %f\n",att);
}
```

OUTPUT:

```
Enter number of process : 3
Enter 3 process times : 3 2 5
Enter 3 process priorites : 6 2 4
Cpu time and priorities before sorting
3       6
2       2
5       4
Cpu times and priorities after sorting
2       2
5       4
3       6
Cpu time        Priority        Waiting time        Turn around time
2                   2                   0                   2
5                   4                   2                   7
3                   6                   7                   10
Average waiting time : 3.000000
Average turn around time : 5.666667
```

VIVA QUESTIONS:

1. What is the drawback of priority scheduling algorithm?
2. Differentiate preemptive and non-preemptive priority algorithm?
3. What is starvation ?

AIM: 1(d) Write a program to implement the Round Robin CPU scheduling algorithm.

DESCRIPTION:

1. The queue structure in ready queue is of First In First Out (FIFO) type.
2. A fixed time is allotted to every process that arrives in the queue. This fixed time is known as time slice or time quantum.
3. The first process that arrives is selected and sent to the processor for execution. If it is not able to complete its execution within the time quantum provided, then an interrupt is generated using an automated timer.
4. The process is then stopped and is sent back at the end of the queue. However, the state is saved and context is thereby stored in memory. This helps the process to resume from the point where it was interrupted.
5. The scheduler selects another process from the ready queue and dispatches it to the processor for its execution. It is executed until the time Quantum does not exceed.
6. The same steps are repeated until all the process are finished.

Different formulas to be calculated in Round Robin:

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time. Waiting Time = Turn Around Time – Burst Time

Steps to find waiting times of all processes:

1. Create an array rem_bt[] to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array)

2. Create another array wt[] to store waiting times of processes. Initialize this array as 0.
3. Initialize time : t = 0
4. Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet.
    a. If rem_bt[i] > quantum
        i. t = t + quantum
        ii. bt_rem[i] -= quantum;
    b. Else // Last cycle for this process
        i. t = t + bt_rem[i];
        ii. wt[i] = t - bt[i]
        iii. bt_rem[i] = 0; // This process is over

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
	char pname[20][10];
	int n,i,bt[20],ptime[20],tq,count=0,tot=0,wt[20],twt=0,ttt=0,e[20];
	float awt,att;
	printf("enter number of process : ");
	scanf("%d",&n);
	for(i=1;i<=n;i++)
	{
		printf("enter %d process name : ",i);
		scanf(" %s",pname[i]);
		printf("enter %d process time : ",i);
		scanf("%d",&ptime[i]);
		bt[i]=ptime[i];
	}
	printf("enter time quantum : ");
	scanf("%d",&tq);
	while(count<n)
	{
			for(i=1;i<=n;i++)
			{
				if(bt[i]!=0)
				{
					if(bt[i]<=tq)
					{
						tot+=bt[i];
```

```
                                        bt[i]=0;
                                        e[i]=tot;
                                        count++;
                                }
                                else
                                {
                                        bt[i]=bt[i]-tq;
                                        tot+=tq;
                                }
                                printf("%s\t%d\n",pname[i],bt[i]);
                        }
                }
        }

        for(i=1;i<=n;i++)
        {
                wt[i]=e[i]-ptime[i];
                twt+=wt[i];
                ttt+=e[i];

        }
        printf("pname\tcpu time\twaiting time\tturn around time");
        for(i=1;i<=n;i++)
        printf("\n%s\t\t%d\t\t%d\t\t%d",pname[i],ptime[i],wt[i],e[i]);
        printf("\ntotal waiting time : ");
        printf("%d",twt);
        awt=(float)twt/n;
        printf("\naverage waiting time : ");
        printf("%f",awt);
        printf("\ntotal turn around time : ");
        printf("%d",ttt);
        att=(float)ttt/n;
        printf("\naverage turn around time : ");
        printf("%f\n",att);

}
```

OUTPUT:

```
Enter number of process : 3
Enter 1 process name : P1
Enter 1 process time : 6
Enter 2 process name : P2
Enter 2 process time : 5
Enter 3 process name : P3
Enter 3 process time : 8
Enter time quantum : 5
P1      1
P2      0
P3      3
```

P1 0
P3 0

Pname	Cpu time	Waiting time	Turn around time
P1	6	10	16
P2	5	5	10
P3	8	11	19

Total waiting time : 26
Average waiting time : 8.666667
Total turn around time : 45
Average turn around time : 15.000000

VIVA QUESTIONS:

1. What are the disadvantages of RR algorithm?
2. List various applications of Round robin algorithm?

EXERCISE: 2

AIM: 2(a) Write a program to implement the MVT memory management technique.

DESCRIPTION:

About MVT (Multiprogramming with a Variable Number of Tasks):

1- Multiprogramming with a Variable Number of Tasks (MVT) is an example of dynamic partitioning.
2- MVT is the memory management technique in which each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system.
3- In variable partitioning scheme there are no partitions at the beginning.
4- Memory is given to the processes as they come.
5- MVT suffers with external fragmentation.
6- MVT is a more efficient user of resources.
7- There is only the OS area and the rest of the available RAM.
8- The memory is allocated to the processes as they enter.
9- This method is more flexible as there is no internal fragmentation and there is no size limitation.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
        int phy,os,size[50],i,n;
        printf("Enter physical memory : ");
        scanf("%d",&phy);
        printf("Enter os memory : ");
        scanf("%d",&os);
        phy=phy-os;
        printf("Enter number of partitions : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter size of process %d : ",i+1);
                scanf("%d",&size[i]);
                if(size[i]<=phy)
```

```
                {
                        phy=phy-size[i];
                        printf("Process %d is allotted\n",i+1);
                }
                else
                        printf("Process %d is blocked\n",i+1);

        }
        printf("Total external fragmentation : %d\n",phy);
}
```

OUTPUT:

```
Enter physical memory : 100
Enter os memory : 20
Enter number of process : 3
Enter size of process-1 : 10
Process 1 is allotted
Enter size of process-2 : 25
Process 2 is allotted
Enter size of process-3 : 50
Process 3 is blocked
Total external fragmentation : 45
```

VIVA QUESTIONS:

1. What is meant by external fragmentation?
2. What is compaction?

AIM: 2(b) Write a program to implement the MFT memory management technique.

DESCRIPTION:

About MFT (Multiprogramming with a Fixed Number of Tasks):

1- Multiprogramming with a Fixed Number of Tasks (MFT) is an example of static partitioning.
2- MFT is one of the old memory management techniques in which the memory is partitioned into fixed size partitions and each job is assigned to a partition.
3- The memory assigned to a partition does not change.
4- The OS is partitioned into fixed sized blocks at the time of installation.
5- It is possible to bind address at the time of compilation.
6- It is not flexible because the number of blocks cannot be changed.
7- There can be memory wastage due to fragmentation.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
        int phy,os,tot,size,i,pm[30],n;
        float inter=0;
        printf("Enter physical memory : ");
        scanf("%d",&phy);
        printf("Enter os memory : ");
        scanf("%d",&os);
        tot=phy-os;
        printf("Enter number of partitions : ");
        scanf("%d",&n);
        size=tot/n;
        for(i=0;i<n;i++)
```

```
        {
                printf("Enter size of process %d : ",i+1);
                scanf("%d",&pm[i]);
                if(pm[i]<=size)
                {
                        printf("Process %d is allotted\n",i+1);
                        inter=inter+(size-pm[i]);
                }
                else

                        printf("Process %d is blocked\n",i+1);
        }
        printf("Total internal fragmentation : %f\n",inter);
}
```

OUTPUT:

```
Enter physical memory : 100
Enter os memory : 20
Enter number of partitions : 3
Enter size of process 1 : 15
Process 1 is allotted
Enter size of process 2 : 10
Process 2 is allotted
Enter size of process 3 : 30
Process 3 is blocked
Total internal fragmentation: 27.000000
```

VIVA QUESTIONS:

1. What is meant by internal fragmentation?
2. what is meant by MFT technique?

EXERCISE: 3

AIM: 3 (a) Write a program to implement the FIFO page replacement algorithm.

DESCRIPTION:

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- We can create a FIFO queue to hold all pages in memory.
- We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

ALGORITHM:

Algorithm for FIFO Page Replacement:

Let capacity be the number of pages that memory can hold.

Let set be the current set of pages in memory.

1. Start traversing the pages.
    a. If set holds less pages than capacity.
        i. Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
        ii. Simultaneously maintain the pages in the queue to perform FIFO.
        iii. Increment page fault
    b. Else

        If current page is present in set, do nothing.

        Else
        i. Remove the first page from the queue as it was the first to be entered in the memory
        ii. Replace the first page in the queue with the current page in the string.
        iii. Store current page in the queue.

iv. Increment page faults.

2. Return page faults.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
        int n,re[20],nf,i,f[20],h,flag1,k,j,count=0;
        printf("enter the number of pages : ");
        scanf("%d",&n);
        printf("enter the page numbers : ");
        for(i=0;i<n;i++)
        scanf("%d",&re[i]);
        printf("enter the number of frames : ");
        scanf("%d",&nf);
        printf("ref. string\tpage frames");
        for(i=0;i<nf;i++)
        {
        f[i]=-1;
        }
        for(i=0;i<n;i++)
        {
     printf("\n");
     printf("%d\t\t",re[i]);
     flag1=0;
     for(k=0;k<nf;k++)
     {
          if(re[i]==f[k])
          {
            flag1=1;
            break;
           }
     }
     if(flag1==0)
     {
                f[j]=re[i];
                j=(j+1)%nf;
                count++;
                for(h=0;h<nf;h++)
                {
                        printf("%d\t",f[h]);
                }

     }
  }
```

```
  printf("\ntotal page faults : %d\n",count);
}
```

OUTPUT:

```
Enter the number of pages : 5
Enter the page numbers : 4 1 2 4 5
Enter the number of frames : 3
Ref. string                Page frames
4                          4      -1     -1
1                          4      1      -1
2                          4      1      2
4
5                          5      1      2
Total page faults : 4
```

VIVA QUESTIONS:

1. What is Belady's Anomaly?
2. What is the need of page replacement?

AIM: 3(b) Write a program to implement the LRU page replacement algorithm

DESCRIPTION:

This algorithm use the recent past as an approximation of the near future, then we can replace then that *has not been used* for the longest period of time.

ALGORITHM:

Let capacity be the number of pages that memory can hold.

Let set be the current set of pages in memory.

1. Start traversing the pages.
    a. If set holds less pages than capacity
        i. Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
        ii. Simultaneously maintain the recent occurred index of each page in a map called indexes.
        iii. Increment page fault
    b. Else
        If current page is present in set, do nothing.

Else

i. Find the page in the set that was least recently used. We find it using index array. We basically need to replace the page with minimum index.

ii. Replace the found page with current page.

iii. Increment page faults.

iv. Update index of current page.

2. Return page faults.

PROGRAM:

```
 #include<stdio.h>
#include<conio.h>
#include<string.h>
 int findmin();
 int i,j,pf=0,f,n,min,ref[40],frame[10],avail,ind=0,count=0,k,time[10];

void main()
{

  printf("enter the number of pages : ");
  scanf("%d",&n);
  printf("enter the page numbers : ");
  for(i=0;i<n;i++)
  scanf("%d",&ref[i]);
  printf("enter the number of frames : ");
  scanf("%d",&f);
  printf("page frames:");
  printf("\n");
        for(i=0;i<f;i++)
        {
                frame[i]=-1;
                time[i]=0;
        }

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

                avail=0;

                for(j=0;j<f;j++)
                {
                        if(frame[j]==ref[i])
                        {
                                avail=1;
                                count++;
```

```
                        time[j]=count;
                        break;
                }
        }
        if(avail==0)
        {
                k=findmin();
                frame[k]=ref[i];
                count++;
                time[k]=count;
                pf++;


                for(j=0;j<f;j++)
                {
                        if(frame[j]!=-1)
                        printf("\t%d",frame[j]);


                }
                printf("\n");
        }
    }
    printf("total page faults : %d\n",pf);

}

int findmin()
{
    min=time[0];
    ind=0;
    for(k=1;k<f;k++)
    {
        if(time[k]<min)
        {
            min=time[k];
            ind=k;
        }
    }
    return ind;
}
```

OUTPUT:

```
Enter the number of pages : 6
Enter the page numbers : 5 7 5 6 7 3
Enter the number of frames : 3
Page frames:
        5
        5       7
        5       7       6
        3       7       6
Total page faults : 4
```

VIVA QUESTIONS:

1. What is the need of page replacement?
2. Differentiate FIFO with LRU page replacement algorithm?

AIM: 3(C) Write a program to implement the Optimal page replacement algorithm.

DESCRIPTION:

- Replace the page that will not be used for the longest period of time.
- Optimal Page Replacement has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.

ALGORITHM:

The idea is simple, for every reference we do following:

- If referred page is already present, increment hit count.
- If not present, find if a page that is never referenced in future. If such a page exists, replace this page with new page. If no such page exists, find a page that is referenced farthest in future. Replace this page with new page.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int findoptimal();
int i,j,pf=0,nf,r,ffree=0,found,np,max,u,v,ref[40],frame[10],avail,ind[10],k;

void main()
{
```

```
        printf("enter the number of pages : ");
        scanf("%d",&np);
        printf("enter the page numbers : ");
        for(i=0;i<np;i++)
        scanf("%d",&ref[i]);
        printf("enter the number of frames : ");
        scanf("%d",&nf);
        printf("ref. string\tpage frames");
        for(i=0;i<nf;i++)
        {
        frame[i]=-1;
        }
        for(i=0;i<np;i++)
        {
        printf("\n");
        printf("%d\t\t",ref[i]);
        avail=0;
        for(j=0;j<nf;j++)
                {
                        if(frame[j]==ref[i])
                        {
                                avail=1;
                        for(j=0;j<nf;j++)
                                printf("%d\t",frame[j]);
                                break;
                        }
                }
                if(avail==0)
                {
                        if(ffree<nf)
                        {
                                k=ffree;
                                ffree++;
                        }
                        else
                                k=findoptimal(i+1);
                        frame[k]=ref[i];
                        pf++;
                        for(j=0;j<nf;j++)
                                printf("%d\t",frame[j]);
                }
        }
        printf("\ntotal page faults : %d\n",pf);

}

int findoptimal(int p)
{
        for(u=0;u<nf;u++)
```

```
        {
                found=0;
                for(v=p;v<np;v++)
                {
                        if(frame[u]==ref[v])
                        {
                                ind[u]=v;
                                found=1;
                                break;
                        }
                }
                if(found==0)
                        return u;
        }
        max=ind[0];
        for(u=1;u<nf;u++)
        {       if(ind[u]>max)
                {
                        max=ind[u];
                        r=u;
                }
        }
        return r;
}
```

OUTPUT:

```
Enter the number of pages : 10
Enter the page numbers : 2 3 4 2 1 3 7 5 4 3
Enter the number of frames : 3
Ref. string             Page frames
2                       2       -1      -1
3                       2       3       -1
4                       2       3       4
2                       2       3       4
1                       1       3       4
3                       1       3       4
7                       7       3       4
5                       5       3       4
4                       5       3       4
3                       5       3       4
Total page faults : 6
```

VIVA QUESTIONS:

1. What is the advantage of OPTIMAL page replacement over FIFO and LRU?

EXERCISE: 4

AIM: Write a program to simulate Bankers algorithm for dead lock avoidance.

DESCRIPTION:

- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- This number may not exceed the total number of resources in the system.
- System has M resources and N processes

Data structures:

- **Available:**
  - A vector of length *m* indicates the number of available resources of each type.
  - If *Available[j]* equals *k,* then *k* instances of resource type *Ri* are available.
- **Max:**
  - An *n* x *m* matrix defines the maximum demand of each process.
  - If *Max[i] [j]* equals *k,* then process P_i may request at most *k* instances of resource type R_j.
- **Allocation:**
  - An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
  - If *Allocation[i][j]* equals k, then process P_i is currently allocated k instances of resource type R_j.
- **Need:**
  - An *n* x *m* matrix indicates the remaining resource need of each process.

- If *Need[i][j]* equals *k,* then process P_i may need *k* more instances of resource type R_j to complete its task. Note that *Need[i][j]* equals *Max[i][j]- Allocation [i][j].*

ALGORITHM:

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length 'm' and 'n' respectively.

   Initialize: Work= Available

   Finish [i]=false; for i=1,2,.....,n

2. Find an i such that both

   a) Finish [i]=false

   b) Need_i<=work

   if no such i exists goto step (4)

3. Work=Work + Allocation_i

   Finish[i]= true

   goto step(2)

4. If Finish[i]=true for all i, then the system is in safe state.

Safe sequence is the sequence in which the processes can be safely executed.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
        int seq[10],m,n,i,j,alloc[10][10],max[10][10],need[10][10]={0},avai[10];
        int finish[10],flag, npleft,c,k=0;
        printf("enter types of resources : ");
        scanf("%d",&m);
        printf("enter available resources instance of %d types resources\n", m);
        for(i=0;i<m;i++)
```

```
scanf("%d",&avai[i]);
printf("enter number of processes : ");
scanf("%d",&n);
printf("enter allocation matrix");
for(i=0;i<n;i++)
{
        for(j=0;j<m;j++)
        {
                scanf("%d",&alloc[i][j]);

        }
}
printf("enter maximum resource matrix\n");
for(i=0;i<n;i++)
{
        for(j=0;j<m;j++)
                scanf("%d",&max[i][j]);
}
printf("the given allocation matrix is\n");
for(i=0;i<n;i++)
{
        for(j=0;j<m;j++)
        {
                printf("%d ",alloc[i][j]);

        }
printf("\n");
}
printf("the need matrix is\n");
for(i=0;i<n;i++)
{
        for(j=0;j<m;j++)
        {
                need[i][j]=max[i][j]-alloc[i][j];
                printf("%d ",need[i][j]);

        }

        printf("\n");
}

printf("the available resource vector is : ");
for(i=0;i<m;i++)
printf("%d ",avai[i]);
for(i=0;i<n;i++)
        finish[i]=0;
npleft=n;
while(npleft!=0)
{
```

```
                for(i=0;i<n;i++)
                {
                   c=0;
                   if(finish[i]==0)
                   {
                        for(j=0;j<m;j++)
                        {
                                if(need[i][j]<=avai[j])
                                        c++;
                        }
                        if(c==m)
                        {

                                finish[i]=1;
                                for(j=0;j<m;j++)
                                        avai[j]=avai[j]+alloc[i][j];
                                seq[k++]=i;
                        }
                    }
                }
                npleft--;
        }
        flag=0;
        for(i=0;i<n;i++)
        {
                if(finish[i]==0)
                        flag=1;
        }
        if(flag==1)
                printf("\nthe system is in unsafe state");
        else
        {
                printf("\nsystem is in safe sate and safe sequence is\n");
                for(i=0;i<n;i++)
                        printf("p - %d\t",seq[i]);
                printf("\n");
        }

}
```

OUTPUT:

```
Enter types of resources : 3
Enter available resources instance of 3 types resources
3 3 2
Enter number of processes : 5
Enter allocation matrix
0 1 0
2 0 0
```

```
3 0 3
2 1 1
0 0 2
Enter maximum resource matrix
7 5 3
3 2 2
9 0 2
2 2 2
1 3 3
The given allocation matrix is
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
The need matrix is
7 4 3
1 2 2
6 0 -1
0 1 1
1 3 1
The available resource vector is : 3 3 2
System is in safe sate and safe sequence is
p - 1  p - 3  p - 4  p - 0  p - 2
```

VIVA QUESTIONS:

1. What are the data structures used in this algorithm?
2. Define safe state?
3. Define safe sequence?
4. Is both deadlock state and unsafe state are same?

EXERCISE: 5

AIM: 5(a) Write a program to implement the FCFS disk scheduling algorithm.

DESCRIPTION:

FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

- The I/O requests are served or processes according to their arrival.
- The request arrives first will be accessed and served first. Since it follows the order of arrival, it causes the wild swings from the innermost to the outermost tracks of the disk and vice versa.
- The farther the location of the request being serviced by the read/write head from its current location, the higher the seek time will be.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int n,i,queue[30],head,diff,seek=0;
        printf("enter number of queue elements : ");
        scanf("%d",&n);
        printf("enter the work queue : ");
        for(i=0;i<n;i++)
        scanf("%d",&queue[i]);
        printf("enter the disk head starting position : ");
        scanf("%d",&head);
        for(i=0;i<n;i++)
        {
                diff=abs(head-queue[i]);
                seek=seek+diff;

                head=queue[i];
        }
        printf("total seek time : %d\n",seek);

}
```

OUTPUT:

```
 Enter number of queue elements : 5
Enter the work queue : 10 50 40 30 20
```

Enter the disk head starting position : 15
Total seek time : 75

VIVA QUESTIONS:

1. How requests are serviced using FCFS algorithm?
2. What is the need of Disk scheduling?
3. Define seek time?
4. Define random access time?
5. Define Rotational latency?

AIM: 5(b) Write a program to implement the SSTF disk scheduling algorithm.

DESCRIPTION:

- In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first.
- So, the seek time of every request is calculated in advance in queue and then they are scheduled according to their calculated seek time.
- As a result, the request near the disk arm will get executed first.
- SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

ALGORITHM:

- Let request array represents an array storing indexes of tracks that have been requested. 'head' is the position of disk head.
- Find the positive distance of all tracks in the request array from head.
- Find a track from requested array which has not been accessed/serviced yet and has minimum distance from head.
- Increments the total seek count with this distance.
- Currently serviced track position now becomes the new head position.
- Go to step 2 until all tracks in request array have not been serviced.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
```

```
        int n,i,queue[30],visit[30],head,j,min,seek=0,pos;
        printf("enter number of queue elements : ");
        scanf("%d",&n);
        printf("enter the work queue : ");
        for(i=0;i<n;i++)
        {
                scanf("%d",&queue[i]);
                visit[i]=0;
        }
        printf("enter the disk head starting position : ");
        scanf("%d",&head);
        for(i=0;i<n;i++)
        {
                min=999;
                for(j=0;j<n;j++)
                {
                        if(visit[j]==0)
                        {
                                if(min>abs(head-queue[j]))
                                {
                                        min=abs(head-queue[j]);
                                        pos=j;
                                }
                        }
                }
                visit[pos]=1;

                head=queue[pos];
                seek=seek+min;
        }
        printf("total seek time : %d\n",seek);


}
```

OUTPUT:

```
Enter number of queue elements : 5
Enter the work queue : 10 50 40 30 20
Enter the disk head starting position : 15
Total seek time : 45
```

VIVA QUESTIONS:

1. How requests are serviced using SSTF scheduling algorithm?
2. Which disk scheduling algorithm is best and why?

AIM: 5(c) Write a program to implement the SCAN disk scheduling algorithm.

DESCRIPTION:

- In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path.
- So, this algorithm works like an elevator and hence also known as elevator algorithm.
- As a result, the requests at the mid-range are serviced more and those arriving behind the disk arm will have to wait.

ALGORITHM:

- The elevator algorithm (also scan) is a disk scheduling algorithm to determine the motion of the disk's arm and head in servicing read and write requests.
- This algorithm is named after the behavior of a building elevator, where the elevator continues to travel in its current direction (up or down) until empty, stopping only to let individuals off or to pick up new individuals heading in the same direction.
- From an implementation perspective, the drive maintains a buffer of pending read/write requests, along with the associated cylinder number of the request.
- When a new request arrives while the drive is idle, the initial arm/head movement will be in the direction of the cylinder where the data is stored, either in or out.
- As additional requests arrive, requests are serviced only in the current direction of arm movement until the arm reaches the edge of the disk.
- When this happens, the direction of the arm reverses, and the requests that were remaining in the opposite direction are serviced, and so on.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int size,n,q[20],visit[20],t,head,pos,seek=0,i,j;
        printf("enter number of queue elements : ");
        scanf("%d",&n);
        printf("enter the work queue : ");
        for(i=0;i<n;i++)
```

```
{
        scanf("%d",&q[i]);
        visit[i]=0;
}

printf("enter the disk head starting position : ");
scanf("%d",&head);
for(i=0;i<n;i++)
{
        for(j=i+1;j<n;j++)
        {
                if(q[i]>q[j])
                {
                        t=q[i];
                        q[i]=q[j];
                        q[j]=t;
                }
        }
}
for(i=n;i>=1;i--)
{
        if(q[i]<head)
        {
                seek=seek+abs(head-q[i]);
                visit[i]=1;
                head=q[i];
        }
}
for(i=0;i<n;i++)
{
        if(visit[i]==0)
        {

                seek=seek+abs(head-0);
                head=0;
                break;
        }
}

for(i=0;i<n;i++)
{
        if(visit[i]==0)
        {
                seek=seek+abs(head-q[i]);

                visit[i]=1;
                head=q[i];
        }
}
```

```
        printf("total seek time : %d\n",seek);


}
```

OUTPUT:

```
Enter number of queue elements : 5
Enter the work queue : 10 50 40 30 20
Enter the disk head starting position : 15
Total seek time : 65
```

VIVA QUESTIONS:

1. Explain about SCAN disk scheduling?
2. What is the other name for SCAN algorithm?

AIM: 5(d) Write a program to implement the C-SCAN disk scheduling algorithm.

DESCRIPTION:

- In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction.
- So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.
- These situations are avoided in C-SCAN algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there.
- So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SC

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int size,n,q[20],visit[20],t,head,pos,seek=0,i,j,max,tot=0;
        float average=0;

        printf("enter the max range of disk : ");
        scanf("%d",&size);
              max=size-1;
        printf("enter the disk head starting position : ");
        scanf("%d",&head);
        printf("enter number of queue elements : ");
```

```
scanf("%d",&n);
printf("enter the work queue : ");
for(i=0;i<n;i++)
{
        scanf("%d",&q[i]);
        visit[i]=0;
}


for(i=0;i<n;i++)
{
        for(j=i+1;j<n;j++)
        {
                if(q[i]>q[j])
                {
                        t=q[i];
                        q[i]=q[j];
                        q[j]=t;
                }
        }
}
for(i=0;i<n;i++)
{
        if(q[i]>head)
        {
                seek=abs(head-q[i]);
                tot=tot+seek;
                printf("disk head moves from %d to %d with seek
                %d\n",head,q[i],seek);

                visit[i]=1;
                head=q[i];
        }
}

                seek=abs(head-size);
                tot=tot+seek;
                printf("disk head moves from %d to %d with seek
                %d\n",head,size,seek);
                head=size;


                seek=abs(size-0);
                tot=tot+seek;
                printf("disk head moves from %d to %d with seek
                %d\n",size,0,seek);
                head=0;

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

```
        {
                if(visit[i]==0)
                {
                        seek=abs(head-q[i]);
                        tot=tot+seek;
                        printf("disk head moves from %d to %d with seek
                        %d\n",head,q[i],seek);
                        visit[i]=1;
                        head=q[i];
                }
        }
        printf("total seek time : %d\n",tot);
        average=(float)tot/n;
        printf("average seek time : %f\n",average);


}
```

OUTPUT:

```
Enter the max range of disk: 200
Enter the disk head starting position: 50
Enter number of queue elements: 8
Enter the work queue: 90 120 35 122 38 128 65 68
Disk head moves from 50 to 65 with seek15
Disk head moves from 65 to 68 with seek 3
Disk head moves from 68 to 90 with seek 22
Disk head moves from 90 to 120 with seek 30
Disk head moves from 120 to 122 with seek 2
Disk head moves from 122 to 128 with seek 6
Disk head moves from 128 to 200 with seek 72
Disk head moves from 200 to 0 with seek 200
Disk head moves from 0 to35 with seek 35
Disk head moves from 35 to 38 with seek 3
Total seek time: 388
Average seek time: 48.500000
```

VIVA QUESTIONS:

1. What is the process of C-SCAN scheduling algorithm?
2. Compare SCAN with C-SCAN?

ADDITIONAL LAB EXPERIMENTS:

1. **AIM:** Write a C program to implement Shortest Job First CPU scheduling algorithm with preemption

DESCRIPTION:

- Preemptive SJF. In Preemptive SJF Scheduling, jobs are put into the ready queue as they come. A process with shortest burst time begins execution.
- If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle

PROGRAM:

```
#include<stdio.h>
struct
{
        int bt,wt,tat,at,ft;
}p[20];
void main()
{
        int n,i,min,nrs,limit,rbt[20],m;
        clrscr();
        printf("enter the no of processses");
        scanf("%d",&n);
         printf("enter the arrival times");
         for(i=1;i<=n;i++)
        {
                printf("enterthe arrvial time %d",i);
                scanf("%d",&p[i].at);
        }
        limit=nrs=p[1].at;
        for(i=1;i<=n;i++)
         {
                printf("enter the burest times  of process %d",i);
                scanf("%d",&p[i].bt);
                limit+=p[i].bt;
                rbt[i]=p[i].bt;
         }
        do
        {
                 min=999;
                 for(i=1;p[i].at<=nrs&&i<=n;i++)
                {
                        if(rbt[i]>0&&min>rbt[i])
                        {
                                m=i;
```

```
                        min=rbt[i];
                    }
                }
    rbt[m]-=1;
    nrs+=1;
            if(rbt[m]==0)
            {
                p[m].ft=nrs;
                p[m].tat=p[m].ft-p[m].at;
                p[m].wt=p[m].tat-p[m].bt;
            }
        }while(nrs<limit);
        printf("processes  \t bt\tft\ttat\twt\n");
        for(i=1;i<=n;i++)
        {
            printf("%d\t%d\t%d\t%d\t%d\n",i,p[i].bt,p[i].ft,p[i].tat,p[i].wt);
        }
  getch();
}
```

OUTPUT:

```
Enter number of processes5
Enter arrival times 2 1 3 4 5
Enter burst times3 6 7 1 2
Enter priorities2 3 4 5 1
Arrival time  Burst time  Waiting time  Priority  Ending time  Turn around time
2       3       0       2       5       3
1       6       6       3       13      12
3       7       10      4       20      17
4       1       16      5       21      17
5       2       0       1       7       2
```

VIVA QUESTIONS:

1. Difference between preemptive SJF and Non-preemptive SJF?
2. Define SRTF?

2. **AIM:** Write a C program to implement Priority CPU scheduling algorithm with preemption

DESCRIPTION:

- In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time.
- The One with the highest priority among all the available processes will be given the CPU next.

PROGRAM:

```
#include<stdio.h>
struct
{
        int bt,wt,tat,at,ft,pt;
}p[20];
void main()
{
        int n,i,min,nrs,limit,rbt[20],m,pr[20];
        clrscr();
        printf("enter the no of process");
        scanf("%d",&n);
        printf("enter the arival times");
        for(i=1;i<=n;i++)
        {
                scanf("%d",&p[i].at);
         }
        limit=nrs=p[1].at;
        printf("enter the priority");
        for(i=1;i<=n;i++)
        {
                scanf("%d",&p[i].pt);
                pr[i]=p[i].pt;
        }

        printf("enter the burest times");
        for(i=1;i<=n;i++)
        {
                scanf("%d",&p[i].bt);
                limit+=p[i].bt;
                rbt[i]=p[i].bt;
        }
        do
         {
                min=999;
                for(i=1;p[i].at<=nrs&&i<=n;i++)
                {
```

```
                        if(rbt[i]>0&&min>pr[i])
                        {
                                m=i;
                                min=pr[i];
                        }
                }
        rbt[m]-=1;
        nrs+=1;
                if(rbt[m]==0)
                {
                        p[m].ft=nrs;
                        p[m].tat=p[m].ft-p[m].at;
                        p[m].wt=p[m].tat-p[m].bt;
                }
         }while(nrs<limit);
         printf("process \t bt \t ft \t tat \t wt \n");
        for(i=1;i<=n;i++)
        {
                printf("%d\t%d\t%d\t%d\t%d\n",i,p[i].bt,p[i].ft,p[i].tat,p[i].wt);
         }
  getch();
}
```

OUTPUT:

```
Enter number of processes5
Enter arrival times3 4 5 1 2
Enter burst times6 7 3 2 1
Arrival time  Burst time  Waiting time Ending time  Turn around time
3       6       6       15      12
4       7       11      22      18
5       3       3       11      6
1       2       5       8       7
2       1       3       6       4
```

VIVA QUESTIONS:

1. Difference between preemptive priority and non-preemptive priority?
2. Define calculation of waiting time and turnaround time?