Content

Ex. No. :1
Date :

Job Scheduling Algorithms

Aim:

To write a c program to implement FCFS and SJF job scheduling techniques.

Concepts Used:

Throughput:

Throughput is the measure of the number of processes that are completed per time unit.

Turnaround time:

From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time:

Waiting time is the sum of the periods spent waiting in the ready queue.

Response time:

In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

Scheduling Algorithms:

1. **First-Come First-Served Scheduling:**
   The process that requests the CPU first is allocated the CPU first.

2. **Shortest-Job-First Scheduling:**
   When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

1. First Comes First Serve Scheduling:

```
#include<stdio.h>
#include<conio.h>

struct process
{
 char name[10];
 int hr,min,sec,burst,wait,arrival,exit;
};

void main()
{
 struct process p[20],temp;
 void read_details_of_process(struct process[],int);
 void print_details_of_process(struct process[],int,int);
 int calculate_waiting_time(struct process[],int);
 int n,total;
 clrscr();
 printf("\nEnter the number of process: ");
 scanf("%d",&n);
 read_details_of_process(p,n);
 total=calculate_waiting_time(p,n);
 print_details_of_process(p,n,total);
 getch();
}

void read_details_of_process(struct process p[],int n)
{
 int i,j;
 printf("\nEnter the details of %d processes: ",n);
 for(i=0;i<n;i++)
 {
  printf("\n\nProcess %d:",(i+1));
  printf("\nEnter process name: ");
  scanf("%s",&p[i].name);
  printf("Enter arrival time: ");
  printf("\n\tEnter Hour: ");
  scanf("%d",&p[i].hr);
  label1:
  if(p[i].hr<=24)
  {
   printf("\tEnter Minute: ");
   scanf("%d",&p[i].min);
   label2:
   if(p[i].min<=60)
```

```
   {
    printf("\tEnter Second: ");
    scanf("%d",&p[i].sec);
    label3:
    if(p[i].sec<=60)
    {
     printf("Enter the burst time(in terms of seconds): ");
     scanf("%d",&p[i].burst);
    }
    else
    {
     printf("Enter seconds <= 60: ");
     scanf("%d",&p[i].sec);
     goto label3;
    }
   }
   else
   {
    printf("Enter Minutes <= 60: ");
    scanf("%d",&p[i].min);
    goto label2;
   }
  }
  else
  {
   printf("Enter hour <= 24: ");
   scanf("%d",&p[i].hr);
   goto label1;
  }
  p[i].arrival=p[i].sec+(p[i].min*60)+(p[i].hr*3600);
 }
}

int calculate_waiting_time(struct process p[],int n)
{
 struct process temp;
 int i,j,total=0,t;
 p[0].exit=p[0].arrival+p[0].burst;
 for(i=0;i<n-1;i++)
 {
  for(j=i+1;j<n;j++)
  {
   if(p[i].arrival>p[j].arrival)
   {
    temp=p[i];
    p[i]=p[j];
```

```
    p[j]=temp;
   }
  }
 }
 for(i=0;i<n;i++)
 {
  if(i==0)
   p[i].wait=0;
  else
   if(p[i].arrival>p[i-1].exit)
   {
    p[i].wait=0;
    p[i].exit=p[i].arrival+p[i].burst;
   }
   else
    if(p[i].arrival>p[i-1].arrival&&p[i].arrival<p[i-1].exit)
    {
     t=p[i].arrival-p[i-1].arrival;
     p[i].wait=p[i-1].wait-t+p[i-1].burst;
     p[i].exit=p[i].arrival+p[i].wait+p[i].burst;
    }
    else
    {
     p[i].wait=p[i-1].wait+p[i-1].burst;
     p[i].exit=p[i].arrival+p[i].wait+p[i].burst;
    }
  total+=p[i].wait;
 }
 return total;
}

void print_details_of_process(struct process p[],int n,int total)
{
 int i,j;
 clrscr();
 printf("\nProcess Name\tArrival Time\tBurst Time\tWaiting Time");
 for(i=0;i<n;i++)
 {
  printf("\n%s\t\t%d:%d:%d\t\t%d\t\t%d",p[i].name,p[i].hr,p[i].min,p[i].sec,p[i].burst,p[i].wait);
 }
 printf("\nTotal Waiting Time: %d",total);
 printf("\nAverage Waiting Time: %0.2f",(total/(n*1.0)));
}
```

Sample Input and Output:

Enter the number of process: 4

Enter the details of 4 processes:

Process 1:
Enter process name: p1
Enter arrival time:
    Enter Hour: 4
    Enter Minute: 10
    Enter Second: 10
Enter the burst time(in terms of seconds): 60

Process 2:
Enter process name: p2
Enter arrival time:
    Enter Hour: 4
    Enter Minute: 10
    Enter Second: 15
Enter the burst time(in terms of seconds): 95

Process 3:
Enter process name: p3
Enter arrival time:
    Enter Hour: 4
    Enter Minute: 10
    Enter Second: 30
Enter the burst time(in terms of seconds): 50

Process 4:
Enter process name: p4
Enter arrival time:
    Enter Hour: 5
    Enter Minute: 12
    Enter Second: 15
Enter the burst time(in terms of seconds): 80

Process Name	Arrival Time	Burst Time	Waiting Time
p1	4:10:10	60	0
p2	4:10:15	95	55
p3	4:10:30	50	135
p4	5:12:15	80	0

Total Waiting Time: 190
Average Waiting Time: 47.50

2. Shortest Job First Scheduling

```
#include<conio.h>

struct process
{
 char name[10];
 int burst,wait;
};

void main()
{
 void read_details_of_process(struct process[],int);
 int calculate_waiting_time(struct process[],int);
 void print_details_of_process(struct process[],int,int);
 struct process p[20];
 int total,n;
 clrscr();
 printf("\nEnter the number of process: ");
 scanf("%d",&n);
 read_details_of_process(p,n);
 total=calculate_waiting_time(p,n);
 print_details_of_process(p,n,total);
 getch();
}

void read_details_of_process(struct process p[],int n)
{
 int i,j;
 printf("\nEnter the details of %d processes: ",n);
 for(i=0;i<n;i++)
 {
  printf("\n\nProcess %d:",(i+1));
  printf("\nEnter process name: ");
  scanf("%s",&p[i].name);
  printf("Enter the burst time: ");
  scanf("%d",&p[i].burst);
 }
}

int calculate_waiting_time(struct process p[],int n)
{
 int i,j,t,total=0;
 struct process temp;
 for(i=0;i<n-1;i++)
 {
```

```
  for(j=i+1;j<n;j++)
  {
   if(p[i].burst>p[j].burst)
   {
    temp=p[i];
    p[i]=p[j];
    p[j]=temp;
   }
  }
 }
 for(i=0;i<n;i++)
 {
  if(i==0)
   p[i].wait=0;
  else
   p[i].wait=p[i-1].wait+p[i-1].burst;
   total+=p[i].wait;
 }
 return total;
}

void print_details_of_process(struct process p[],int n,int total)
{
 int i;
 clrscr();
 printf("\nProcess Name\tBurst Time\tWaiting Time");
 for(i=0;i<n;i++)
 {
  printf("\n%s\t\t%d\t\t%d",p[i].name,p[i].burst,p[i].wait);
 }
  printf("\nTotal Waiting Time: %d",total);
  printf("\nAverage Waiting Time: %0.2f",(total/(n*1.0)));
}
```

Sample Input and Output:

Enter the number of process: 4

Enter the details of 4 processes:

Process 1:
Enter process name: p1
Enter the burst time: 60

Process 2:
Enter process name: p2
Enter the burst time: 35

Process 3:
Enter process name: p3
Enter the burst time: 15

Process 4:
Enter process name: p4
Enter the burst time: 75

Process Name	Burst Time	Waiting Time
p3	15	0
p2	35	15
p1	60	50
p4	75	110

Total Waiting Time: 175
Average Waiting Time: 43.75

Result:

Thus, First Comes First Served Scheduling and Shortest Job First Scheduling algorithms have been implemented in C language and tested for various sample inputs.

Ex. No. :2
Date :

Disk Scheduling Algorithms

Aim:

To write c programs to implement FCFS and SSTF, disk scheduling techniques

Concepts Used:

First Come First Served Scheduling:

The FCFS algorithm selects the request based on the first come arrival basis.

Shortest-Seek-Time-First Scheduling:

The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

First-Come First-Served Scheduling

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

void main()
{
 int i,sum=0,n,st;
 int a[20],b[20],dd[20];
 clrscr();

 do
 {
  printf("\nEnter the block number between 0 and 200: ");
  scanf("%d",&st);
 }while((st>=200)||(st<0));

 printf("\nOur disk head is on the %d block",st);
 a[0]=st;
 printf("\nEnter the no. of request: ");
 scanf("%d",&n);
 printf("\nEnter request: ");
```

```
 for(i=1;i<=n;i++)
 {
  printf("\nEnter %d request: ",i);
  scanf("%d",&a[i]);
  do
  {
   if((a[i]>200)||(a[i]<0))
   {
    printf("\nBlock number must be between 0 and 200!");
   }
  }while((a[i]>200)||(a[i]<0));
 }

 for(i=0;i<=n;i++)
  dd[i]=a[i];
 printf("\n\t\tFIRST COME FIRST SERVE: ");
 printf("\nDISK QUEUE:");

 for(i=0;i<=n;i++)
  printf("\t%d",a[i]);
 printf("\n\nACCESS ORDER:");

 for(i=0;i<=n;i++)
 {
  printf("\t%d",dd[i]);
  if(i!=n)
   sum+=abs(dd[i]-dd[i+1]);
 }
 printf("\n\nTotal no. of head movements: %d",sum);
 getch();
}
```

Sample Input and Output:

Enter the block number between 0 and 200: 53

Our disk head is on the 53 block
Enter the no. of request: 8

Enter request:

Enter 1 request: 98

Enter 2 request: 183

Enter 3 request: 37

Enter 4 request: 122

Enter 5 request: 14

Enter 6 request: 124

Enter 7 request: 65

Enter 8 request: 67

First Come First Served:

Disk Queue:	53	98	183	37	122	14	124	65	67
Access Order:	53	98	183	37	122	14	124	65	67

Total no. of head movements: 640

Shortest-Seek-Time-First Scheduling

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

void main()
{
 int i,j,z,sum=0,c=0,n,n1,st,min;
 int a[20],b[20],dd[20];
 clrscr();

 do
 {
  printf("\nEnter the block number between 0 and 200: ");
  scanf("%d",&st);
 }while((st>=200)||(st<0));
 printf("\nOur disk head is on the %d block",st);
 a[0]=st;
 printf("\nEnter the no. of request: ");
 scanf("%d",&n);
 printf("\nEnter request: ");

 for(i=1;i<=n;i++)
 {
  printf("\nEnter %d request: ",i);
  scanf("%d",&a[i]);
```

```
do
 {
  if((a[i]>200)||(a[i]<0))
  {
   printf("\nBlock number must be between 0 and 200!");
  }
 }while((a[i]>200)||(a[i]<0));
}

for(i=0;i<=n;i++)
 dd[i]=a[i];
n1=n;
b[0]=dd[0];
st=dd[0];

while(n1>0)
{
 j=1;
 min=abs(dd[0]-dd[1]);
 for(i=2;i<n1+1;i++)
 {
  if(abs(st-dd[i])<=min)
  {
   min=abs(st-dd[i]);
  j=i;
  }
 }
 c++;
 b[c]=dd[j];
 st=dd[j];
 dd[0]=dd[j];
 --n1;

 for(z=j;z<n1+1;z++)
  dd[z]=dd[z+1];
 dd[z]='\0';
}
printf("\n\t\tSHORTEST SEEK TIME FIRST: ");
printf("\nDISK QUEUE:");
for(i=0;i<=n;i++)
 printf("\t%d",a[i]);
printf("\n\nACCESS ORDER:");
for(i=0;i<=c;i++)
{
printf("\t%d",b[i]);
if(i!=c)
```

```
  sum+=abs(b[i]-b[i+1]);
 }
 printf("\n\nTotal no. of head movements: %d",sum);
 getch();
}
```

Sample Input and Output:

Enter the block number between 0 and 200: 53

Our disk head is on the 53 block
Enter the no. of request: 8

Enter request:
Enter 1 request: 98

Enter 2 request: 183

Enter 3 request: 37

Enter 4 request: 122

Enter 5 request: 14

Enter 6 request: 124

Enter 7 request: 65

Enter 8 request: 67

Shortest Seek Time First:

Disk Queue: 53 98 183 37 122 14 124 65 67

Access Order: 53 65 67 37 14 98 122 124 183

Total no. of head movements: 236

Result:

Thus, C programs to implement different disk scheduling techniques have been written successfully and tested with various samples.

Ex. No. :3
Date :

Memory Allocation Techniques

Aim:

To write C programs to implement First Fit, Best Fit, and Worst Fit memory allocation techniques.

Concepts Used:

1. **First fit.** Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
2. **Best fit.** Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
3. **Worst fit.** Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

1) First Fit

```
#include<stdio.h>
#include<conio.h>

int next=0,f1,p,c,l,sum;
int asize[30],fsize[30],f1size[30],bsize[30];

void main()
{
 char ch;
 int blsize,i,k;
 void first_fit(int);
 clrscr();
 printf("\nEnter the number of free block: ");
 scanf("%d",&f1);
 sum=50;
 printf("\nEnter within the width 480.");
 printf("\nEnter width within the limit.\n");
 for(i=0;i<f1;i++)
 {
  printf("Enter the size of the block%d: ",i);
  scanf("%d",&fsize[i]);
  if(fsize[i]>481)
  {
```

```
   printf("\nExceeding the limit, re-enter the value!");
   continue;
  }
  f1size[i]=fsize[i];
  asize[i]=0;
  sum=sum+fsize[i];
 }
 printf("\nEnter the number of process: ");
 scanf("%d",&p);
 for(i=0;i<p;i++)
 {
  printf("Enter the size of allocated memory process%d: ",i);
  scanf("%d",&bsize[i]);
 }
 for(i=0;i<p;i++)
 first_fit(bsize[i]);
 getch();
}

void first_fit(int n)
{
 int k=0,i,s1;
 for(i=0;i<f1;i++)
 {
  if(fsize[i]>=n)
  {
   asize[i]=asize[i]+n;
   next=i+1;
   printf("\n\nMEMORY ALLOCATION IN BLOCK:%d ",i);
   s1=50;
   for(l=0;l<i;l++)
    s1=s1+fsize[l]+asize[l];
   printf("\nMemory allocated for process:%d ",asize[l]);
   fsize[i]=-n;
   k=1;
   break;
  }
 }
 if(k==0)
  printf("\n\nNo matching block for %d \n",n);
 }
```

Sample Input and Output:

Enter the number of free block: 3
Enter within the width 480.

Enter width within the limit.

Enter the size of the block 0: 100
Enter the size of the block 1: 50
Enter the size of the block 2: 200

Enter the number of process: 3

Enter the size of allocated memory process 0: 200
Enter the size of allocated memory process 1: 250
Enter the size of allocated memory process 2: 100

Memory Allocation in Block:2
Memory allocated for process:200

No matching block for 250

Memory Allocation in Block:0
Memory allocated for process:100

2) Best Fit

```
#include<stdio.h>
#include<conio.h>

int next=0,f1,p,c,l,sum;
int asize[30],fsize[30],f1size[30],bsize[30];

void main()
{
 char ch;
 int blsize,i,k;
 void best_fit(int);
 clrscr();
 printf("\nEnter the number of free block: ");
 scanf("%d",&f1);
 sum=50;
 printf("\nEnter within the width 480.");
 printf("\nEnter width within the limit.\n");
 for(i=0;i<f1;i++)
 {
```

```
  printf("Enter the size of the block%d: ",i);
  scanf("%d",&fsize[i]);
  if(fsize[i]>481)
  {
   printf("\nExceeding the limit, re-enter the value!");
   continue;
  }
  f1size[i]=fsize[i];
  asize[i]=0;
  sum=sum+fsize[i];
 }
 printf("\nEnter the number of process: ");
 scanf("%d",&p);
 for(i=0;i<p;i++)
 {
  printf("Enter the size of allocated memory process%d: ",i);
  scanf("%d",&bsize[i]);
 }
 for(i=0;i<p;i++)
 best_fit(bsize[i]);
 getch();
}

void best_fit(int n)
{
 int l,s,k=0,i,s1;
 int min1=10000;
 for(i=0;i<=f1;i++)
 {
  if((fsize[i]-n)>=0)
  {
   s=fsize[i]-n;
   if(s<min1)
   {
    min1=s;
    k=i+1;
   }
  }
 }
 if(k!=0)
 {
  next=k;
  fsize[k-1]=min1;
  asize[k-1]+=n;
  s1=50;
```

```
  for(l=0;l<k-1;l++)
   s1=s1+fsize[l]+asize[l];
  printf("\n\nMemory allocated in block:%d",(k-1));
  printf("\nMemory allocated for process:%d",asize[l]);
 }
 else
  printf("\n\nNo matching block for %d",n);
}
```

Sample Input and Output:

Enter the number of free block: 3

Enter within the width 480.
Enter width within the limit.

Enter the size of the block 0: 100
Enter the size of the block 1: 50
Enter the size of the block 2: 200

Enter the number of process: 3

Enter the size of allocated memory process 0: 200
Enter the size of allocated memory process 1: 250
Enter the size of allocated memory process 2: 100

Memory allocated in block: 2
Memory allocated for process: 200

No matching block for 250

Memory allocated in block: 0
Memory allocated for process: 100

3) Worst Fit

```
#include<stdio.h>
#include<conio.h>

int next=0,f1,p,c,l,sum;
int asize[30],fsize[30],f1size[30],bsize[30];

void main()
{
 char ch;
 int blsize,i,k;
```

```
 void worst_fit(int);
 clrscr();
 printf("\nEnter the number of free block: ");
 scanf("%d",&f1);
 sum=50;
 printf("\nEnter within the width 480.");
 printf("\nEnter width within the limit.\n");
 for(i=0;i<f1;i++)
 {
  printf("Enter the size of the block%d: ",i);
  scanf("%d",&fsize[i]);
  if(fsize[i]>481)
  {
   printf("\nExceeding the limit, re-enter the value!");
   continue;
  }
  f1size[i]=fsize[i];
  asize[i]=0;
  sum=sum+fsize[i];
 }
 printf("\nEnter the number of process: ");
 scanf("%d",&p);
 for(i=0;i<p;i++)
 {
  printf("Enter the size of allocated memory process%d: ",i);
  scanf("%d",&bsize[i]);
 }
 for(i=0;i<p;i++)
 worst_fit(bsize[i]);
 getch();
}

void worst_fit(int n)
{
 int l,s,k=0,i,s1;
 int max1=0;
 for(i=0;i<f1;i++)
 {
  if((fsize[i]-n)>0)
  {
   s=fsize[i]-n;
   if(s>=max1)
   {
    max1=s;
    k=i+1;
   }
```

```
  }
 }
 if(k!=0)
 {
  next=k;
  fsize[k-1]=max1;
  asize[k-1]=asize[k-1]+n;
  s1=50;
  for(l=0;l<k-1;l++)
   s1+=fsize[l]+asize[l];
  printf("\n\nMemory allocated in block:%d",(k-1));
  printf("\nMemory allocated for process:%d",asize[l]);
 }
 else
  printf("\n\nNo matching block for %d",n);
}
```

Sample Input and Output:

Enter the number of free block: 3

Enter within the width 480.
Enter width within the limit.

Enter the size of the block 0: 100
Enter the size of the block 1: 50
Enter the size of the block 2: 200

Enter the number of process: 3

Enter the size of allocated memory process 0: 200
Enter the size of allocated memory process 1: 250
Enter the size of allocated memory process 2: 100
No matching block for 200

No matching block for 250

Memory allocated in block: 2
Memory allocated for process: 100

Result:

Thus, C programs to implement different Memory Allocation techniques have been written successfully and tested with various samples.

Ex. No. :4
Date :

Memory Management using Paging

Aim:

To write a C program to implement the paging technique

Concepts Used:

• Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.

• The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.

• Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table.

• The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

• The size of a page is typically a power of 2.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>

void main()
{
 int page_size,log_size,phy_size,no_pages_log,no_frames_phy;
 int i,j,check,x,frame_no,frame_alloc[50];
 int check_pow_2(int);
 char log_content[50][10],phy_content[50][10];;
 clrscr();

 lab:

 printf("\n\nEnter the page size(power of 2): ");
 scanf("%d",&page_size);  check=check_pow_2(page_size);
 if(check==0)
 {
  printf("\nEnter page size as power of 2.");
```

```
 goto lab;
 }

 lab1:

 printf("\n\nEnter the logical memory size(power of 2): ");
 scanf("%d",&log_size);
 check=check_pow_2(log_size);
 if(check==0)
 {
  printf("\nEnter logical memory size as power of 2.");
  goto lab1;
 }
 no_pages_log=log_size/page_size;
 printf("\nNo. of pages in logical memory: %d",no_pages_log);

 lab2:

 printf("\n\nEnter the physical memory size(power of 2): ");
 scanf("%d",&phy_size);
 check=check_pow_2(phy_size);
 if(check==0||phy_size<log_size)
 {
  printf("\nEnter physical memory size as power of 2 and \ngreater than or equal to logical
memory size.");
  goto lab2;
 }
 no_frames_phy=phy_size/page_size;
 printf("\nNo. of frames in physical memory: %d",no_frames_phy);
 printf("\n\nEnter the contents of logical memory: ");
 x=0;
 for(i=0;i<no_pages_log;i++)
 {
  for(j=0;j<page_size;j++)
  {
   scanf("%s",&log_content[x]);
   x++;
  }
 }
 clrscr();
 x=0;
 printf("\nLOGICAL MEMORY: ");
 for(i=0;i<no_pages_log;i++)
 {
  printf("\nPAGE%d: ",i);
  j=0;
```

```
 for(j=0;j<page_size;j++)
 {
  printf("\n\tLogical address %d: %s",x,log_content[x]);
  x++;
 }
}
getch();
clrscr();
for(i=0;i<no_frames_phy;i++)
{
 frame_alloc[i]=0;
}
x=0;
for(i=0;i<no_pages_log;i++)
{

 lab3:

 printf("\nEnter frame no for page %d(0-%d): ",i,no_frames_phy);
 scanf("%d",&frame_no);
 if(frame_no>=no_frames_phy)
 {
  printf("\n%d frame is not available.Enter another frame no.",frame_no);
  goto lab3;
 }
 if(frame_alloc[frame_no]==0)
 {
  for(j=0;j<page_size;j++)
  {
   strcpy(phy_content[(frame_no*page_size)+j],log_content[(i*page_size)+j]);
   x++;
  }
  frame_alloc[frame_no]=1;
 }
 else
 {
  printf("\n%d frame is already allocated. Enter another frame.",frame_no);
  goto lab3;
 }
}
for(i=0;i<no_frames_phy;i++)
{
 if(frame_alloc[i]==0)
 {
  for(j=0;j<page_size;j++)
  {
```

```
    strcpy(phy_content[(i*page_size)+j],"-");
   }
  }
 }
 clrscr();
 x=0;
 printf("\nPHYSICAL MEMORY.");
 for(i=0;i<no_frames_phy;i++)
 {
  printf("\nFRAME%d: ",i);
  for(j=0;j<page_size;j++)
  {
   printf("\n\tPhysical Address%d: %s",x,phy_content[(i*page_size)+j]);
   x++;
  }
  printf("\nPress any key.");
  getch();
 }
 getch();
}

int check_pow_2(int n)
{
 int i=1,flag=0,j;
 while(1)
 {
  j=pow(2,i);
  if(j==n)
  {
   flag=1;
   break;
  }
  if(i==n/2)
   break;
  i++;
 }
 if(flag==1)
  return 1;
 else
  return 0;
}
```

Sample Input and Output:

Enter the page size(power of 2): 4

Enter the logical memory size(power of 2): 16

No. of pages in logical memory: 4

Enter the physical memory size(power of 2): 32

No. of frames in physical memory: 8

Enter the contents of logical memory:
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p

Logical Memory:

PAGE 0:

Logical address 0: a
Logical address 1: b
Logical address 2: c
Logical address 3: d

PAGE 1:

Logical address 4: e
Logical address 5: f
Logical address 6: g
Logical address 7: h

PAGE 2:

Logical address 8: i
Logical address 9: j
Logical address 10: k
Logical address 11: l

PAGE 3:

Logical address 12: m
Logical address 13: n
Logical address 14: o
Logical address 15: p

Enter frame no for page 0(0-8): 5
Enter frame no for page 1(0-8): 6
Enter frame no for page 2(0-8): 1
Enter frame no for page 3(0-8): 2

Physical Memory:

FRAME 0:

Physical Address 0: -
Physical Address 1: -
Physical Address 2: -
Physical Address 3: -
Press any key.

FRAME 1:

Physical Address 4: i
Physical Address 5: j
Physical Address 6: k
Physical Address 7: l
Press any key.

FRAME 2:

Physical Address 8: m
Physical Address 9: n
Physical Address 10: o
Physical Address 11: p
Press any key.

FRAME 3:

Physical Address 12: -
Physical Address 13: -
Physical Address 14: -
Physical Address 15: -
Press any key.

FRAME 4:

Physical Address 16: -
Physical Address 17: -
Physical Address 18: -
Physical Address 19: -
Press any key.

FRAME 5:

Physical Address 20: a
Physical Address 21: b
Physical Address 22: c
Physical Address 23: d

FRAME 6:

Physical Address 24: e
Physical Address 25: f
Physical Address 26: g
Physical Address 27: h
Press any key.

FRAME 7:

Physical Address 28: -
Physical Address 29: -
Physical Address 30: -
Physical Address 31: -
Press any key

Result:

Thus, the C program to implement the paging memory management technique has been written successfully and tested with various samples.

Ex. No. :5
Date :

Memory Management using Segmentation

Aim:

To write a C program to implement segmentation memory management scheme

Theory:

- Segmentation is a memory-management scheme that supports this user view of memory.
- A logical address space is a collection of segments. Each segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.
- The segment number is used as an index to the segment table.
- The offset of the logical address must be between 0 and the segment limit.

Program:

```
#include<stdio.h>
#include<conio.h>

void main()
{
 int no_seg_log,limit[20],base_phy[20],end_phy[20];
 int tem[40],i,j;
 clrscr();
 printf("\nEnter the no. of segments in logical memory: ");
 scanf("%d",&no_seg_log);
 for(i=0;i<no_seg_log;i++)
 {
  printf("\nEnter limit of segment%d: ",i);
  scanf("%d",&limit[i]);
  printf("\nEnter base address of segment%d: ",i);
  scanf("%d",&base_phy[i]);
  end_phy[i]=base_phy[i]+limit[i];
 }
  printf("\nSEGMENT\tLIMIT\tRANGE");
 for(i=0;i<no_seg_log;i++)
 {
  printf("\n%d\t%d\t%d - %d",i,limit[i],base_phy[i],end_phy[i]);
 }
 getch();
}
```

Sample Input and Output:

Enter the no. of segments in logical memory : 5

Enter limit of segment 0 : 1000

Enter base address of segment 0 : 1400

Enter limit of segment 1 : 400

Enter base address of segment 1 : 6300

Enter limit of segment 2 : 400

Enter base address of segment 2 : 4300

Enter limit of segment 3 : 1100

Enter base address of segment 3 : 3200

Enter limit of segment 4 : 1000

Enter base address of segment 4 : 4700

SEGMENT	LIMIT	RANGE
0	1000	1400 - 2400
1	400	6300 - 6700
2	400	4300 - 4700
3	1100	3200 - 4300
4	1000	4700 – 5700

Result:

Thus, the C program to implement segmentation memory management scheme has been written successfully and tested with various samples.

Ex. No. :6
Date :

Banker's Safety Algorithm

Aim:

To display the present state of the system is Safe or Unsafe by implementing banker algorithm.

Concept:

Deadlock is a situation where in two or more competing actions are waiting f or the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether them allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data Structure Used:

- **Available:** A vector of length m indicates the number of available resources of each type. If Available[j] equals k, there are k instances of resource type R_j available.

- **Max:** An n x m matrix defines the maximum demand of each process. If M[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].

Algorithms:

i. **Safety Algorithm:**

1. Let Work and Finish be vectors of length m and n, respectively.
   Initialize Work = Available and Fnish[i] = false for i = 0 , 1 , ..., n - l .
2. Find an i such that both
   Finish[i] ==false
   Need <= Work
   If no such i exists, go to step 4.
3. Work = Work + Allocation,
   Finish[i] = true
   Go to step 2.
4. If Finish[i] == true for all. i, then the system is in a safe state.

ii. Resource Request Algorithm:

1. If $Request_i$ <= $Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i$ <= Available, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

   Available = Available - $Request_i$

   $Allocation_i$ = $Allocation_i$ + $Request_i$

   $Need_i$ = $Need_i$ - $Request_i$
4. If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i, must wait for $Request_i$, and the old resource-allocation state is restored.

Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int allocated[15][15],max[15][15],need[15][15],avail[15],tres[15],work[15],flag[15];
 int pno,rno,i,j,prc,count,t,total;
 count=0;
 clrscr();
 printf("\n Enter number of process:");
 scanf("%d",&pno);
 printf("\n Enter number of resources:");
 scanf("%d",&rno);
 for(i=1;i<=pno;i++)
 {
  flag[i]=0;
 }
 printf("\n Enter total numbers of each resources:");
 for(i=1;i<=rno;i++)
  scanf("%d",&tres[i]);
 printf("\n Enter Max resources for each process:");
 for(i=1;i<=pno;i++)
 {
  printf("\n for process %d:",i);
  for(j=1;j<= rno;j++)
   scanf("%d",&max[i][j]);
 }
 printf("\n Enter allocated resources for each process:");
```

```
for(i=1;i<= pno;i++)
{
 printf("\n for process %d:",i);
 for(j=1;j<= rno;j++)
  scanf("%d",&allocated[i][j]);
}
printf("\n available resources:\n");
for(j=1;j<= rno;j++)
{
 avail[j]=0;
 total=0;
 for(i=1;i<= pno;i++)
 {
  total+=allocated[i][j];
 }
 avail[j]=tres[j]-total;
 work[j]=avail[j];
 printf(" %d \t",work[j]);
}
do
{
 for(i=1;i<= pno;i++)
 {
  for(j=1;j<= rno;j++)
  {
   need[i][j]=max[i][j]-allocated[i][j];
  }
 }
 printf("\n Allocated matrix Max need");
 for(i=1;i<= pno;i++)
 {
  printf("\n");
  for(j=1;j<= rno;j++)
  {
   printf("%4d",allocated[i][j]);
  }
  printf("|");
  for(j=1;j<= rno;j++)
  {
   printf("%4d",max[i][j]);
  }
  printf("|");
  for(j=1;j<= rno;j++)
  {
   printf("%4d",need[i][j]);
  }
```

```
 }
 prc=0;
 for(i=1;i<= pno;i++)
 {
  if(flag[i]==0)
  {
   prc=i;
   for(j=1;j<= rno;j++)
   {
    if(work[j]< need[i][j])
    {
     prc=0;
     break;
    }
   }
  }
  if(prc!=0)
   break;
 }
 if(prc!=0)
 {
  printf("\n Process %d completed",i);
  count++;
  printf("\n Available Resources:");
  for(j=1;j<= rno;j++)
  {
   work[j]+=allocated[prc][j];
   allocated[prc][j]=0;
   max[prc][j]=0;
   flag[prc]=1;
   printf(" %d",work[j]);
  }
 }
}while(count!=pno&&prc!=0);
if(count==pno)
 printf("\nThe system is in a Safe State!!");
else
 printf("\nThe system is in an Unsafe State!!");
getch();
}
```

Sample Input/ Output:

Enter number of process:5
Enter number of resources:3
Enter total numbers of each resources:10 5 7
Enter Max resources for each process:
for process 1:7 5 3

for process 2:3 2 2

for process 3:9 0 2

for process 4:2 2 2

for process 5:4 3 3

Enter allocated resources for each process:
for process 1:0 1 0

for process 2:3 0 2

for process 3:3 0 2

for process 4:2 1 1

for process 5:0 0 2

Available Resources:
2 3 0

Allocated	Max	Need
0 1 0	7 5 3	7 4 3
3 0 2	3 2 2	0 2 0
3 0 2	9 0 2	6 0 0
2 1 1	2 2 2	0 1 1
0 0 2	4 3 3	4 3 1

Process 2 completed
Available Resources: 5 3 2

Allocated	Max	Need
0 1 0	7 5 3	7 4 3
0 0 0	0 0 0	0 0 0
3 0 2	9 0 2	6 0 0
2 1 1	2 2 2	0 1 1
0 0 2	4 3 3	4 3 1

Process 4 completed
Available Resources: 7 4 3

Allocated	Max	Need
0 1 0	7 5 3	7 4 3
0 0 0	0 0 0	0 0 0
3 0 2	9 0 2	6 0 0
0 0 0	0 0 0	0 0 0
0 0 2	4 3 3	4 3 1

Process 1 completed
Available Resources: 7 5 3

Allocated	Max	Need
0 0 0	0 0 0	0 0 0
0 0 0	0 0 0	0 0 0
3 0 2	9 0 2	6 0 0
0 0 0	0 0 0	0 0 0
0 0 2	4 3 3	4 3 1

Process 3 completed
Available Resources: 10 5 5

Allocated	Max	Need
0 0 0	0 0 0	0 0 0
0 0 0	0 0 0	0 0 0
0 0 0	0 0 0	0 0 0
0 0 0	0 0 0	0 0 0
0 0 2	4 3 3	4 3 1

Process 5 completed
Available Resources: 10 5 7

The system is in a safe state!!

Case 2:

Enter Number of Process: 5

Enter Number of Resources: 3
Enter total number of each resources: 10 5 7
Enter Max resources for each Process:
for process 1: 7 5 3

for process 2: 3 2 2

for process 3: 9 0 2

for process 4: 2 2 2

for process 5: 4 3 3

Enter allocated resources for each process:
for process 1: 0 3 0

for process 2: 3 0 2

for process 3: 3 0 2

for process 4: 2 1 1

for process 5: 0 0 2

Available Resources:
2 1 0

Allocated	Max	Need
0 3 0	7 5 3	7 2 3
3 0 2	3 2 2	0 2 0
3 0 2	9 0 2	6 0 0
2 1 1	2 2 2	0 1 1
0 0 2	4 3 3	4 3 1

The System is in an Unsafe State!!

Result:

Thus, a C program to display the present state of the system in Safe or Unsafe by implementing the banker algorithm is written and tested with various cases.

Ex. No. :7
Date :

Dining Philosopher Problem

Aim:

To write a C program to implement the Dining Philosopher Problem.

Concept:

Consider eight philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by eight chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with eight single chopsticks. When a philosopher thinks, he does not interact with his colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him. A philosopher cannot pick up a chopstick that is already in the hand of a neighbour. When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks. When he is finished eating, he puts down both of his chopsticks and starts thinking again. A Philosopher may pick up his chopsticks only if both of them are available in order to avoid deadlock.

Program:

```
/* Dining Philosopher */

#include<stdio.h>
#include<conio.h>
#include<string.h>

char state[10];
void pickup(int);
void test(int);
void putdown(int);
void print_status();
char pname[10][10];
char hun[10];
int no_phil,max_eater;

void main()
{
 int i,j,k,n,pos,no_eat,round=1;
 char c;
 clrscr();
 printf("\nEnter number of philosophers: ");
 scanf("%d",&no_phil);
 max_eater=no_phil/2;
 printf("\n%d philosophers can eat at a time to avoid deadlock.",max_eater);
 printf("\nEnter %d philosopher's names one by one: ",no_phil);
 for(i=0;i<no_phil;i++)
```

```
{
 scanf("%s",pname[i]);
}
for(i=0;i<no_phil;i++)
 state[i]='t';
for(i=0;i<no_phil;i++)
{
 printf("\nposition %d:%s",i,pname[i]);
}
getch();
while(1)
{
 clrscr();
 printf("\nROUND%d",round);
 printf("\n-------");
 printf("\nstatus: ");
 print_status();
 no_eat=0;
 for(j=0;j<no_phil;j++)
 {
  if(state[j]=='h')
  {
   pickup(j);
   if(state[j]=='e')
    no_eat++;
  }
 }
 printf("\nEnter %d philosophers who wants to eat: ",(max_eater-no_eat));
 for(i=0;i<(max_eater-no_eat);i++)
 {
  lab:
   printf("\n\nEnter hungry philosopher%d: ",(i+1));
   scanf("%s",hun);
   for(j=0;j<no_phil;j++)
   {
    k=strcmp(pname[j],hun);
    if(k==0)
    {
     pos=j;
     break;
    }
   }
   pickup(pos);
   if(state[pos]=='h')
    goto lab;
 }
```

```
  getch();
  clrscr();
  printf("\nCurrent status: ");
  print_status();
  for(j=0;j<no_phil;j++)
  {
   if(state[j]=='e')
   {
    putdown(j);
   }
  }
  printf("\nDo you want to continue?(y/n): ");
  c=getch();
  if(c=='n'||c=='N')
   break;
  else
   round++;
 }
 getch();
}

void pickup(int i)
{
 state[i]='h';
 test(i);
}

void print_status()
{
 int i;
 printf("\nPHILOSOPHER\tSTATE");
 for(i=0;i<no_phil;i++)
 {
  printf("\n%s\t\t%c",pname[i],state[i]);
 }
}

void test(int i)
{
 if((state[(i+(no_phil-1))%no_phil]!='e')&&(state[i]=='h')&&(state[(i+1)%no_phil]!='e'))
 {
  state[i]='e';
 }
 if(state[i]!='e')
  printf("\n%s must wait since her neighbour is eating",pname[i]);
 else
```

```
  if(state[i]=='e')
   printf("\nHungry philosopher %s is granted to eat",pname[i]);
}

void putdown(int i)
{
 state[i]='t';
}
```

Sample Input/ Output:

```
Enter number of philosophers: 8
4 philosophers can eat at a time to avoid deadlock.
Enter 8 philosopher's names one by one:
a
b
c
d
e
f
g
h

position 0:a
position 1:b
position 2:c
position 3:d
position 4:e
position 5:f
position 6:g
position 7:h

ROUND1
-------
status:

PHILOSOPHER             STATE
        a                       t
        b                       t
        c                       t
        d                       t
        e                       t
        f                       t
        g                       t
        h                       t
Enter 4 philosophers who wants to eat:
```

Enter hungry philosopher1: a
Hungry philosopher a is granted to eat
Enter hungry philosopher2: c
Hungry philosopher c is granted to eat

Enter hungry philosopher3: d
d must wait since her neighbour is eating

Enter hungry philosopher3: e
Hungry philosopher e is granted to eat

Enter hungry philosopher4: g
Hungry philosopher g is granted to eat

Current status:

PHILOSOPHER	STATE
a	e
b	t
c	e
d	h
e	e
f	t
g	e
h	t

Do you want to continue?(y/n): y

ROUND2
---
status:

PHILOSOPHER	STATE
a	t
b	t
c	t
d	h
e	t
f	t
g	t
h	t

Hungry philosopher d is granted to eat

Enter 3 philosophers who wants to eat:

Enter hungry philosopher1: e
e must wait since her neighbour is eating

Enter hungry philosopher1: f
Hungry philosopher f is granted to eat

Enter hungry philosopher2: h
Hungry philosopher h is granted to eat

Enter hungry philosopher3: b
Hungry philosopher b is granted to eat

Current status:

PHILOSOPHER	STATE
a	t
b	e
c	t
d	e
e	h
f	e
g	t
h	e

Do you want to continue?(y/n):n

Result:

Thus, a C program to implement the Dining Philosopher Problem is written and tested with various inputs.