Government Engineering College, Aurangabad

Laboratory Manual

Operating System

For

Third Year CSE Students

Dept: Computer Science & Engineering (NBA Accredited)

LABORATORY MANUAL CONTENTS

This manual is intended for the Third year students of CS & Engineering branch in the subject of Operating System. This manual typically contains practical/Lab Sessions related Operating System covering various aspects related the subject to enhanced understanding.

Although, as per the syllabus, Operating System examples are prescribed, we have made the efforts to cover various aspects of Operating System

Students are advised to thoroughly go through this manual rather than only topics mentioned in the syllabus as practical aspects are the key to understanding and conceptual visualization of theoretical aspects covered in the books.

Good Luck for your Enjoyable Laboratory Sessions

Mr. Vivek Kshirsagar
Head of Department

Computer Science & Engg. Dept.

Mr. Vivek Kshirsagar Miss. Karishma Dule

Lecturer	Practical Incharge (M.E. Stud.)
Computer Science & Engg. Dept.	Computer Science & Engg. Dept.

DOs and DON'Ts in Laboratory:

1. Make entry in the Log Book as soon as you enter the Laboratory.

2. All the students should sit according to their roll numbers starting from their left to right.

3. All the students are supposed to enter the terminal number in the log book.

4. Do not change the terminal on which you are working.

5. All the students are expected to get at least the algorithm of the program/concept to be implemented.

6. Strictly observe the instructions given by the teacher/Lab Instructor.

Instruction for Laboratory Teachers:

1. Submission related to whatever lab work has been completed should be done during the next lab session. The immediate arrangements for printouts related to submission on the day of practical assignments.

2. Students should be taught for taking the printouts under the observation of lab teacher.

3. The promptness of submission should be encouraged by way of marking and evaluation patterns that will benefit the sincere students.

Practical List

1) To implement Page Replacement algorithm for First In First Out.

2) To implement CPU & scheduling algorithm for First come First serve Scheduling.

3) To implement CPU scheduling algorithm for shortest job first Scheduling.

4) To implement CPU scheduling for Round Robin Scheduling.

5) To implement a ‘C’ program to perform priority scheduling.

6) To implement Producer-Consumer Problem.

7) To implement Banker Algorithm.

8) To implement Encryption Algorithm.

9) To implement Dining philosophers Problem.

10) To Study Memory Management.

11)

Practical 1:

Aim: To implement Page Replacement algorithm for First In First Out.

Theory: Page replacement algorithm is an algorithm decides which pages should be writing to disk when new page needs to allocated. Page replacement increases the system performance. While there is also a possibility of picking a randomize page to remove during the page fault occurs, in this case system performance will be optimize if small sized pages is chosen rather than the heavily pages. Page replacement algorithms does the work on the basis of both theoretical and implementations. For example in case of web server, the web server keeps tracks of numbers of pages in the memory cache. A new page is referenced when the memory cache is full. In this case web server decides which page is to be removed.

Page replacement takes the following procedure. If no frame is free, we find one that is not currently being used and free it. We can free the frame by making changing in the page table to point that the page is no longer available in the memory. Now we can modify and use the page fault procedure to include page replacement.

1) Look out the location of the page on the memory disk.

2) Look for a free frame

    - If free frame is available, use it.

    - Use page replacement algorithm to pick the victim page in case of no free Frame.

    - Write the victim page to the memory disk.
      Also make changes the entries in the frame and page table.

3) Read the particular page into newly allocated free frame, change the frame and page tables accordingly.

4) Repeat the whole user process.

There are many different page replacement algorithms. Every operating system probably has its own replacement scheme. Further we are going to discuss various types of algorithms of page replacement algorithms.

FIFO (First-In, First Out):

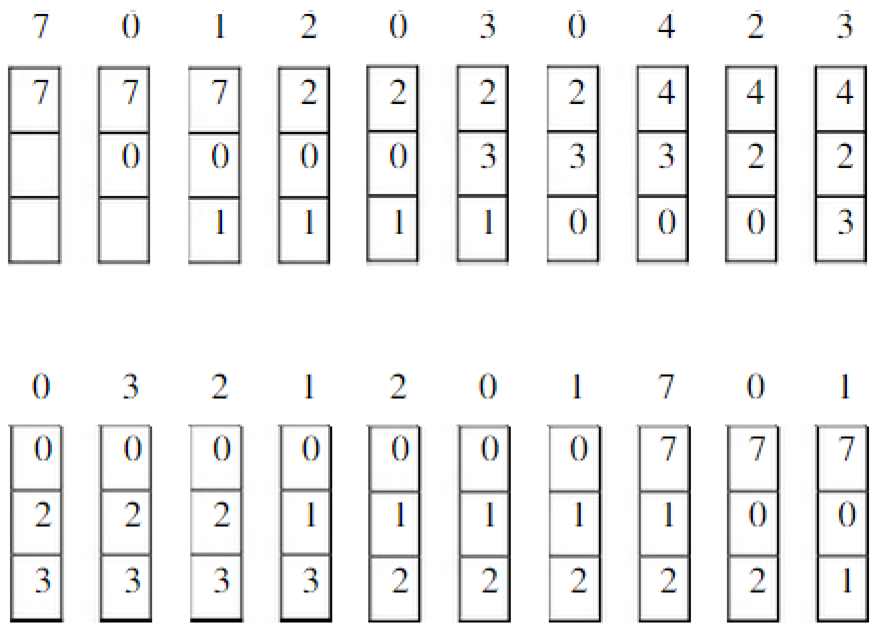
The simplest and low-overhead page replacement algorithm is the FIFO (First-In, First Out) algorithm. A FIFO replacement algorithm links with each page the time when that page was added into the memory; the oldest page is chosen when a page is going to be replaced. We can create a FIFO queue to hold all the pages present in the memory disk. At the head of the queue we replace the page. We insert page at the tail of the queue when a page is added in into the memory disk.

Example:

We consider the following reference string:

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Buffer size: 3



Total page fault occurs: 15

Initially the three frames are empty. The first three references(7, 0, 1) causes pages faults and are added into the three empty frames. The next reference is (2) which replaces the page (7) because page (7) was added in first. Since (0) is the next reference but (0) is already in the memory, so we have no page fault for this reference. This process continues as shown above. Every time a page fault occurs, we show which pages are in our three frames. There are total 15 page faults. The FIFO page replacement algorithm is easy to implement and understand but the performance wise is bad. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and constant use.

Program:

```
#include<stdio.h>

#include<conio.h>

void main()

{

int n,i,a[50],no,frame[10],k,j,avail,cnt=0;

float pr;

clrscr();

printf("\nEnter number of pages:");

scanf("%d",&n);

printf("\nEnter page numbers:");

for(i=1;i<=n;i++)

scanf("%d",&a[i]);

printf("\nEnter number of frames:");

scanf("%d",&no);

for(i=0;i<no;i++)

{

frame[i]=-1;

}

j=0;

printf("\tReferntial String\tPage frames\n");
```

```
        for(i=1;i<=n;i++)
        {
        printf("\t%d\t",a[i]);
        avail=0;
        for(k=0;k<no;k++)
        if(frame[k]==a[i])
        avail=1;
        if(avail==0)
        {
        frame[j]=a[i];
        j=(j+1)%no;
        cnt++;
        for(k=0;k<no; k++)
        printf("\t %d\t", frame[k]);
        }
        printf("\n ");
        }
        printf("\nPage fault=%d",cnt);
        pr=(float)cnt/(float)n;
        printf ("\nPage Rate=%f",pr);
        getch();
}
```

Conclusion: Thus the FIFO program was executed and verified successfully..

Practical 2:

Aim :- To implement CPU & scheduling algorithm for First come First serve scheduling.

Theory:-

First come, first served (FCFS) is an operating system process scheduling algorithm and a network routing management mechanism that automatically executes queued requests and processes by the order of their arrival. With first come, first served, what comes first is handled first; the next request in line will be executed once the one before it is complete.

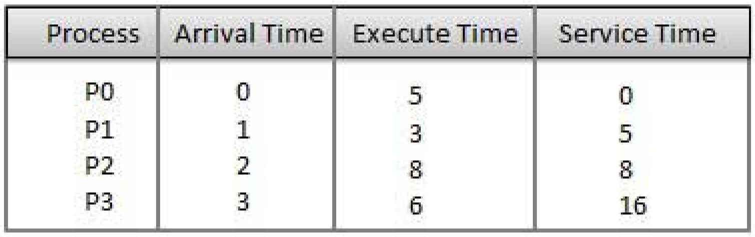
FCFS is also known as first-in, first-out (FIFO) and first come, first choice (FCFC).

FCFS provides an efficient, simple and error-free process scheduling algorithm that saves valuable CPU resources. It uses non preemptive scheduling in which a process is automatically queued and processing occurs according to an incoming request or process order. FCFS derives its concept from real-life customer service.

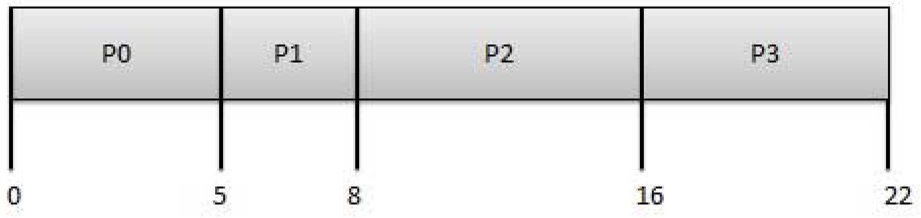
Let's take a look at how FCFS process scheduling works.

Suppose there are three processes in a queue: P1, P2 and P3. P1 is placed in the processing register with a waiting time of zero seconds and 10 seconds for complete processing. The next process, P2, must wait 10 seconds and is placed in the processing cycle until P1 is processed. Assuming that P2 will take 15 seconds to complete, the final process, P3, must wait 25 seconds to be processed. FCFS may not be the fastest process scheduling algorithm, as it does not check for priorities associated with processes. These priorities may depend on the processes' individual execution times.

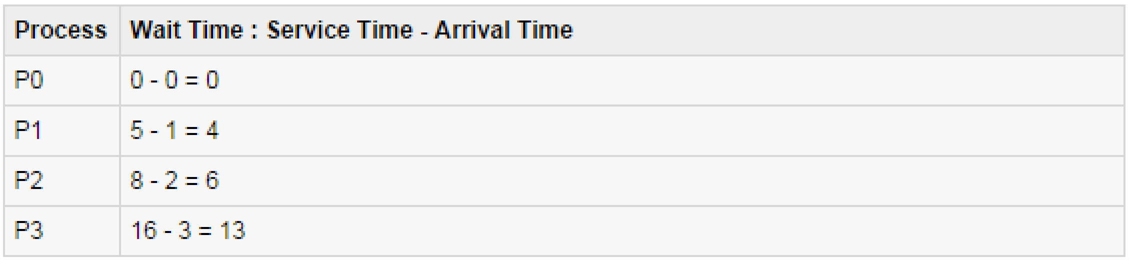
Example:



Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is following



Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.55

Advantage:

- Easy to implement.
- It is very simple.

Disadvantage:

- Troublesome for time sharing systems.
- The average waiting time is very high, this may impact on the performance of the CPU.

Program:

```
#include<stdio.h>
#include<conio.h>
struct process
{
int burst,wait;
}p[20]={0,0};
int main()
{
int n,i,totalwait=0,totalturn=0;
printf("\nEnter The No Of Process :");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter The Burst Time (in ms) For Process #%2d :",i+1);
scanf("%d",&p[i].burst);
}
printf("\nProcess\t Waiting Time TurnAround Time ");
printf("\n\t (in ms) (in ms)");
for(i=0;i<n;i++)
{
printf("\nProcess # %-12d%-15d%-15d",i+1,p[i].wait,p[i].wait+p[i].burst);
p[i+1].wait=p[i].wait+p[i].burst;
totalwait=totalwait+p[i].wait;
totalturn=totalturn+p[i].wait+p[i].burst;
}
printf("\n\nAVERAGE\n---------");
printf("\nWaiting Time : %f ms",totalwait/(float)n);
printf("\nTurnAround Time : %f ms\n\n",totalturn/(float)n);
return 0;
}
```

Conclusion: Thus the FCFS program was executed and verified successfully.

Practical 3:

Aim: To implement CPU scheduling algorithm for shortest job first scheduling.

Theory :-

Shortest job next (SJN), also known as Shortest Job First (SJF) or Shortest Process Next (SPN), is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non preemptive algorithm. Shortest remaining time is a preemptive variant of SJN. Shortest job next is advantageous because of its simplicity and because it minimizes the average amount of time each process has to wait until its execution is complete. However, it has the potential for process starvation for processes which will require a long time to complete if short processes are continually added. Highest response ratio next is similar but provides a solution to this problem. Shortest job next can be effectively used with interactive processes which generally follow a pattern of alternating between waiting for a command and executing it. If the execution burst of a process is regarded as a separate "job", past behaviour can indicate which process to run next, based on an estimate of its running time. Shortest job next is used in specialized environments where accurate estimates of running time are available. Estimating the running time of queued processes is sometimes done using a technique called aging**.**

Advantages:

- Minimizes average waiting time.
- Provably optimal w.r.t. average turnaround time
- Throughput is high.

Disadvantages:

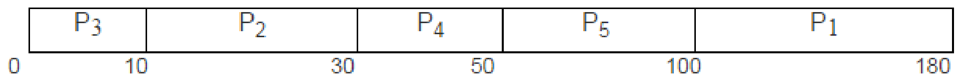
- In general, cannot be implemented.
  - Requires future knowledge
  - In practice, can't actually predict the length of next burst
- Can lead to unfairness or starvation
  - It may penalize processes with high service time requests. If the ready list is saturated, then processes with large service times tend to be left in the ready list while small processes receive service. In extreme case, where the system has little idle time,

processes with large service times will never be served. This total starvation of large processes may be a serious liability of this algorithm.

Example:

PROCESS	ARRIVAL TIME	BURST TIME
P1	0	80
P2	0	20
P3	0	10
P4	0	20
P5	0	50

Gantt chart



PROCESS	WAIT TIME	TURN AROUND TIME
P1	100	180
P2	10	30
P3	0	10
P4	30	50
P5	50	100

Total Wait Time
100 + 10 + 0 + 30 + 50 = 190 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)
190/5 = 38 ms

Total Turn Around Time
180 + 30 + 10 + 50 + 100 = 370 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)
370/5 = 74 ms

Throughput
5 jobs/180 sec = 0.2778 jobs/sec

Program:

```
#include<stdio.h>
struct process
{
int burst,wait,no;
}
p[20]={0,0};
int main()
{
int n,i,j,totalwait=0,totalturn=0;
printf("\nEnter The No Of Process :");
scanf("%d",&n);
for(i=0;i<n;i++){
printf("Enter The Burst Time (in ms) For Process #%2d:",i+1);
scanf("%d",&p[i].burst);
p[i].no=i+1;
}
for(i=0;i<n;i++)
for(j=0;j<n-i-1;j++)
if(p[j].burst>p[j+1].burst)
{
p[j].burst^=p[j+1].burst^=p[j].burst^=p[j+1].burst;
p[j].no^=p[j+1].no^=p[j].no^=p[j+1].no;
}
printf("\nProcess\t Waiting Time TurnAroundTime ");
for(i=0;i<n;i++)
{
printf("\nProcess# %-12d%-15d%-15d",p[i].no,p[i].wait,p[i].wait+p[i].burst);
p[i+1].wait=p[i].wait+p[i].burst;
totalwait=totalwait+p[i].wait;
totalturn=totalturn+p[i].wait+p[i].burst;
}
printf("\n\nAverage\n---------");
printf("\nWaiting Time : %f ms",totalwait/(float)n);
printf("\nTurnAround Time : %f ms\n\n",totalturn/(float)n);
return 0;
}
```

Conclusion: Thus the SJF program was executed and verified successfully.

Practical 4:

Aim: To implement CPU scheduling for Round Robin Scheduling.

Theory:

Round robin is the scheduling algorithm used by the CPU during execution of the process . Round robin is designed specifically for time sharing systems . It is similar to first come first serve scheduling algorithm but the preemption is the added functionality to switch between the processes .

A small unit of time also known as time slice or quantum is set/defined . The ready queue works like circular queue .All processes in this algorithm are kept in the circular queue also known as ready queue . Each New process is added to the tail of the ready/circular queue .

By using this algorithm , CPU makes sure, time slices are assigned to each process in equal portions and in circular order , dealing with all process without any priority .

It is also known as cyclic executive .

- ✓ **Pseudo Code :**

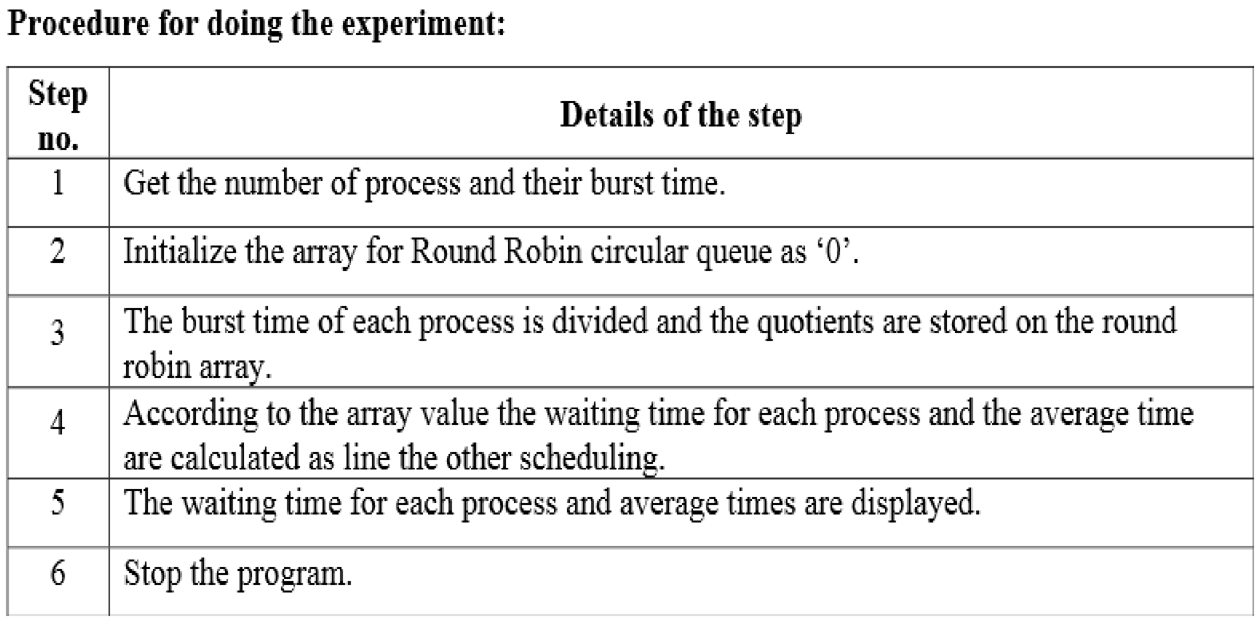
```
1. CPU scheduler picks the process from the circular/ready queue , set a
timer to interrupt it after 1 time slice / quantum and dispatches it .

2. If process has burst time less than 1 time slice/quantum
   a) Process will leave the CPU after the completion
   b) CPU will proceed with the next process in the ready queue / circular
queue.
```

✓ else If process has burst time longer than 1 time slice/quantum

Timer will be stopped . It cause interruption to the OS .
Executed process is then placed at the tail of the circular / ready querue by applying the context

switch
CPU scheduler then proceeds by selecting the next process in the ready queue .

Here , User can calculate the average turnaround time and average waiting time along with the starting and finishing time of each process.



Procedure for doing the experiment:

Step no.	Details of the step
1	Get the number of process and their burst time.
2	Initialize the array for Round Robin circular queue as ‘0’.
3	The burst time of each process is divided and the quotients are stored on the round robin array.
4	According to the array value the waiting time for each process and the average time are calculated as line the other scheduling.
5	The waiting time for each process and average times are displayed.
6	Stop the program.

Program:

```
#include<stdio.h>

struct process

{

int burst,wait,comp,f;

}p[20]={0,0};

int main(){
```

```
int
n,i,j,totalwait=0,totalturn=0,quantum,flag=1,
time=0;
printf("\nEnter The No Of Process :");
scanf("%d",&n);
printf("\nEnter The Quantum time (in ms):");
scanf("%d",&quantum);
for(i=0;i<n;i++)
{
printf("Enter The Burst Time (in ms) ForProcess #%2d :",i+1);
scanf("%d",&p[i].burst);
p[i].f=1;
}
printf("\nOrder Of Execution\n");
printf("\nProcess Starting Ending Remaining");
printf("\nTime Time Time");
while(flag==1)
{
flag=0;
for(i=0;i<n;i++)
{
if(p[i].f==1)
{
flag=1;
j=quantum;
if((p[i].burst-p[i].comp)>quantum)
{
p[i].comp+=quantum;
}
else
{
p[i].wait=time-p[i].comp;
j=p[i].burst-p[i].comp;
p[i].comp=p[i].burst;
p[i].f=0;
}
printf("\nprocess                    #                    %-3d                    %-10d
%-10d%-10d",i+1,time,time+j,p[i].burst-p[i].comp);
time+=j;
}
}
}
printf("\n\n------------------");
printf("\nProcess\t Waiting TimeTurnAround Time ");
for(i=0;i<n;i++)
{
printf("\nProcess # %-12d%-15d%-15d",i+1,p[i].wait,p[i].wait+p[i].burst);
totalwait=totalwait+p[i].wait;
totalturn=totalturn+p[i].wait+p[i].burst;
}
printf("\n\nAverage\n------------------");
printf("\nWaiting Time: %fms",totalwait/(float)n);
printf("\nTurnAround Time : %f ms\n\n",totalturn/(float)n);
return 0;
}
```

Conclusion: Thus the Round Robin program was executed and verified successfully.

Practical 5:

Aim: To write a ‘C’ program to perform priority scheduling.

Theory:

In this scheduling policy the processes are given certain priorities usually specified as a number. They are sorted according to the priorities and the process with highest priority is scheduled first.

This method is quite same as the SJF but the difference is that instead of choosing the next process to work on by the shortest burst time, CPU chooses the next process by the shortest priority value. Here, all the processes are given a priority value. The process with the shortest (The most shortest is 1) priority will be worked on first and so on. Now consider a CPU and also consider a list in which the processes are listed as follows,

Arrival	Process	Burst Time	Priority
0	1	3	2
1	2	2	1
2	3	1	3

Here in Priority Scheduling, what will happen is as follows,

AT 0s:

There is only 1 job, that is Process-1 with Burst Time 3 and Priority 2. So CPU will do 1 second job of Process-1. Thus Process-1 has 2s more job to be done.

AT 1s:

Now there are 2 jobs,

Process-1 that arrived at 0s and has 2s job to be done with Priority 2.

Process-2 that arrived at 1s and has 2s job to be done with Priority 1.

Here, Priority of the job Process-2 is higher (Lower value of priority is higher, that is shortest priority value is higher), So 1s job of Process-2 will be done. Thus process-2 has more 1s job.

AT 2s:

Now there are 3 jobs,

Process-1 that arrived at 0s and has 2s job to be done with Priority 2.

Process-2 that arrived at 1s and has 1s job to be done with Priority 1.

Process-3 that arrived at 2s and has 1s job to be done with Priority 3.

Here, Process-2 has the highest priority, thus CPU will do 1s job of Process-2. So process-2 has more 0s job.

AT 3s:

Now there are 2 jobs,

Process-1 that arrived at 0s and has 2s job to be done with Priority 2.

Process-3 that arrived at 2s and has 1s job to be done with Priority 3.

Here, as Process-1 is with the highest priority (which is 2) so CPU will do 1s job of Process-1. So process-1 has more 1s job.

AT 4s:

There are 2 jobs,

Process-1 that arrived at 0s and has 1s job to be done with Priority 2.

Process-3 that arrived at 2s and has 1s job to be done with Priority 3.

Again, CPU will do 1s job of Process-1 as it is with the highest priority. Thus Process-1 has 0s more job to be done.

AT 5s:

There is only 1 job, that is Process-3 with burst time 1 and priority 3. So CPU will do 1s job of Process-3. Thus Process-3 has 0s more job to be done.

Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities or qualities to compute priority of a process.

Examples of Internal priorities are

- Time limits.
- Memory requirements.
- File requirements,
  for example, number of open files.
- CPU Vs I/O requirements.

Externally defined priorities are set by criteria that are external to operating system such as

- The importance of process.
- Type or amount of funds being paid for computer use.
- The department sponsoring the work.
- Politics.

Priority scheduling can be either preemptive or non preemptive

- A preemptive priority algorithm will preemptive the CPU if the priority of the newly arrival process is higher than the priority of the currently running process.
- A non-preemptive priority algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling is indefinite blocking or starvation.

Program:

```
#include<stdio.h>
#include<conio.h>
struct priority
{
int
num,at,bt,wt,pt; };
struct priority
a[20],temp; int i,j,n;
float sum1=0,sum2=0;
void main()
{
void read();
void sort();
void process();
void print();
clrscr();
read();
sort();
process();
print();
}
void read()
{
printf("enter no.of jobs \n");
scanf("%d",&n);
printf("enter priority burst time\n");
for(i=1;i<=n;i++)
{
printf("job(%d)",i);
scanf("%d%d",&a[i].pt,&a[i].bt);
a[i].num=i;
}
}
void sort()
{
for(i=1;i<=n-1;i++)
{
```

```
for(j=i+1;j<=n;j++)
{
if(a[i].pt>a[j].pt)
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
printf("jobs priority burst_time\n");
for(i=1;i<=n;i++)
{
printf("\njob[%d]\t",a[i].num);
printf("\t%d\t%d\n",a[i].pt,a[i].bt);
}
}
void process()
{
int t=0;
printf("the gantt is \n");
for(i=1;i<(n*10);i++);
printf("-");
printf("\n");
for(i=1;i<=n;i++)
printf("\job%d\t",a[i].num);
printf("\n");
for(i=1;i<(n*10);i++);
printf("-");
printf("\n");
for(i=1;i<=n;i++)
{
a[i].wt=t;
printf("%d\t",t);
t=t+a[i].bt;
}
printf("%d\n",t);

}
void print()
{
```

```
int i;
printf("jobs waiting time t.a.t\n");
printf("_____ _______ _____");
for(i=1;i<=n;i++)
{
printf("\njob[%d]\t%d\t%d",a[i].num,a[i].wt,a[i].wt+a[i].bt);
sum1+=a[i].wt;
sum2+=a[i].wt+a[i].bt;
}
printf("avg w.t=%f\n",(float)sum1/n);
printf("avg t.a.t =%f\n",(float)sum2/n);
}
```

Conclusion: Thus the Priority program was executed and verified successfully.

Practical 6:

Aim: To implement Producer-Consumer Problem.

Theory:

Producer-Consumer problem (also known as the bounded-buffer problem) is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to go to sleep if the buffer is full. The next time the consumer removes an item from the buffer, it wakes up the producer who starts to fill the buffer again. In the same way, the consumer goes to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

The problem can also be generalized to have multiple producers and consumers.

Using semaphores:

Semaphores solve the problem of lost wakeup calls. In the solution below we use two semaphores, fillCount and emptyCount, to solve the problem. fillCount is incremented and emptyCount decremented when a new item has been put into the buffer. If the producer tries to decrement emptyCount while its value is zero, the producer is put to sleep. The next time an item is consumed, emptyCount is incremented and the producer wakes up. The consumer works analogously.

```
Semaphore fillCount = 0
Semaphore emptyCount = BUFFER_SIZE

Procedure producer() {
While (true) {
        item = produceItem()
        down(emptyCount)
        putItemIntoBuffer(item)
        up(fillCount)
}
}
Procedure consumer() {
while(true) {
        down(fillCount)
        item = removeItemFromBuffer()
        up(emptyCount)
        consumeItem(item)
}}
```

The solution above works fine when there is only one producer and consumer. Unfortunately, with multiple producers or consumers this solution contains a serious race condition that could result in two or more processes reading or writing into the same slot at the same time. To understand how this is possible, imagine how the procedure putItemIntoBuffer()can be implemented. It could contain two actions, one determining the next available slot and the other writing into it. If the procedure can be executed concurrently by multiple producers, then the following scenario is possible:

1. Two producers decrement empty Count.

2. One of the producers determines the next empty slot in the buffer.

3. Second producer determines the next empty slot and gets the same result as the first producer.

4. Both producers write into the same slot.

To overcome this problem ,we need a way to make sure that only one producer is executing putItemIntoBuffer() at a time. In other words we need a way to execute a critical section with mutual exclusion. To accomplish this we use a binary semaphore called mutex. Since the value of a binary semaphore can be only either one or zero, only one process can be executing between down(mutex) and up(mutex). The solution for multiple producers and consumers is shown below.

```
Semaphore mutex = 1
Semaphore fillCount = 0
Semaphore emptyCount = BUFFER_SIZE
Procedure producer() {
While (true) {
     item = produceItem()
     down(emptyCount)
     down(mutex)
     putItemIntoBuffer(item)
     up(mutex)
     up(fillCount)
}
     up(fillCount) //the consumer may not finish before the
     producer.
}
Procedure consumer() {
     While (true) {
          down(fillCount)
          down(mutex)
          item = removeItemFromBuffer()
          up(mutex)
          up(emptyCount)
     consumeItem(item)

     }
     }
```

Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int ch,n,c1=0,c2=0,produce[23],consume[23];
 clrscr();
 printf("\n\n\n\n\n\t\n\n\t\t\tEnter Stack Size :  ",n);
 scanf("%d",&n);
 while(1)
 {
  clrscr();
  printf("\t\tProducer Stack (Stack Size : %d
                      )\n\t\t~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~",n);
  display(c1,produce);
  printf("\n\n\t\tConsumer Stack (Stack Size : %d
                      )\n\t\t~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~",n);
  display(c2,consume);
  printf("\n\t\tCHOICES\n\t\t~~~~~~~\n\t1.Producer\n\t2.Consumer\n\t3.
                                                Exit\nEnter your choice :  ");
  scanf("%d",&ch);
  switch(ch)
  {
   case 1:
         if(c1==n)
          printf("Produer stack is FULL.So Producer goes to SLEEP\n");
         else
         {
         c1++;
         printf("\t\tEnter PRODUCE item is :");
         scanf("%d",&produce[c1]);
         }
         break;

   case 2:
         if(c2==n)
            printf("Consumer Stack is FULL.So it goes to SLEEP!..........\n\tReset the
Consumer Stack\n",c2=0);
         else if(c1==0)
           printf("\tProducer stack is EMPTY\n");
         else
          {
        c2++;
          consume[c2]=produce[c1];
          printf("\t\tCONSUME one item");
          c1--;
          }
         break;
```

```
   case 3:
          exit(0);

   default:
          printf("\tIt is Wrong choice,Please enter correct
                                                   choice!............\n");
   }
  getch();
 }
}

display(int c,int stack[])
{
 int i;
 printf("\n-----------------------------------------------------------------------
                                                            -----\n");
 if(c==0)
 printf("\tStack is EMPTY\n\t\t(Now It is sleeping)");
 else
 for(i=1;i<=c;i++)
 printf("\t%d",stack[i]);
 printf("\n-----------------------------------------------------------------------
                                                            -----\n");
 }
```

Conclusion: Thus the Producer-Consumer Problem was executed and verified successfully.

Practical 8:

Aim: To implement Banker Algorithm.

Theory: **Banker's Algorithm**

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of pre-determined maximum possible amounts of all resources, and then makes a "safe-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue. The Banker's algorithm is run by the operating system whenever a process requests resources. The algorithm prevents deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

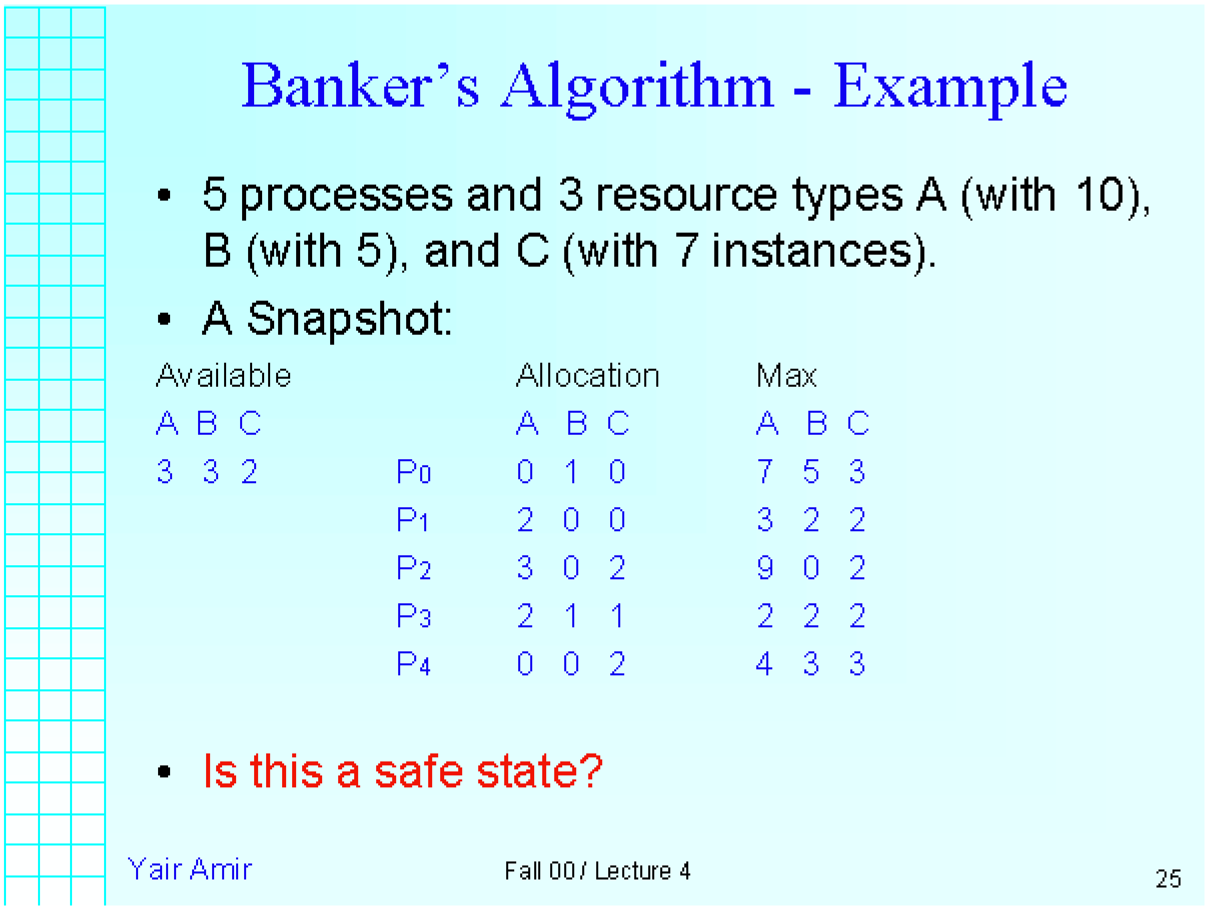
Resources:

For the Banker's algorithm to work, it needs to know three things:

1. How much of each resource each process could possibly request.
2. How much of each resource each process is currently holding.
3. How much of each resource the system has available.

Safe and Unsafe States:

A state is considered safe if it is possible for all processes to finish executing (terminate). Since the system cannot know when a process will terminate, or how many resources it will have requested by then, the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward. This is a reasonable assumption in most cases since the system is not particularly concerned with how long each process runs (at least not from a deadlock avoidance perspective). Also, if a process terminates without acquiring its maximum resources, it only makes it easier on the system. Given that assumption, the algorithm determines if a state is safe by trying to find a hypothetical set of requests by the processes that would allow each to acquire (one-by-one) its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an unsafe state. In a safe state, at least one process should be able to acquire its maximum possible set of resources, and proceed to termination.



Banker's Algorithm - Example

- 5 processes and 3 resource types A (with 10), B (with 5), and C (with 7 instances).
- A Snapshot:

Available		Allocation	Max
A B C		A B C	A B C
3 3 2	P_0	0 1 0	7 5 3
	P_1	2 0 0	3 2 2
	P_2	3 0 2	9 0 2
	P_3	2 1 1	2 2 2
	P_4	0 0 2	4 3 3

- Is this a safe state?

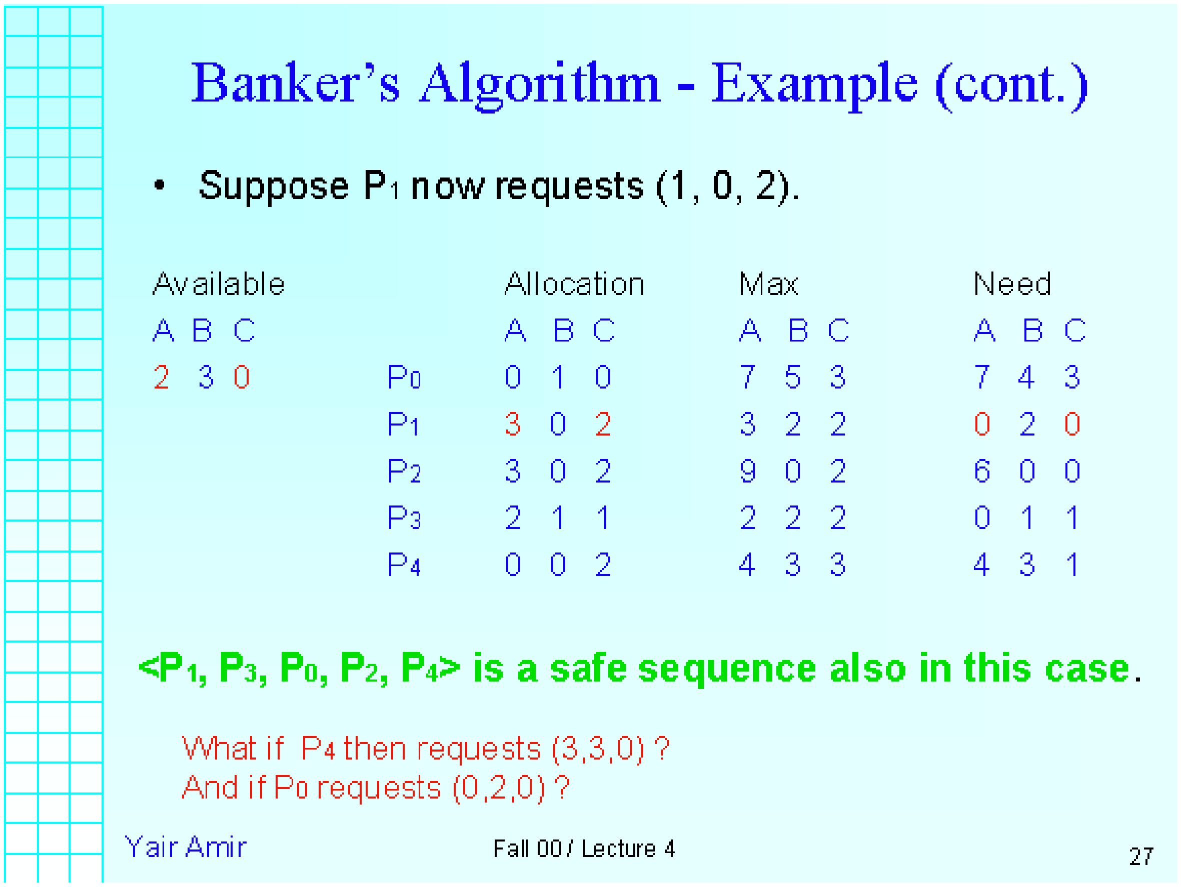
Yair Amir Fall 00 / Lecture 4 25

Banker's Algorithm - Example

- 5 processes and 3 resource types A (with 10), B (with 5), and C (with 7 instances).
- A Snapshot:

Available		Allocation	Max	Need
A B C		A B C	A B C	A B C
3 3 2	P_0	0 1 0	7 5 3	7 4 3
	P_1	2 0 0	3 2 2	1 2 2
	P_2	3 0 2	9 0 2	6 0 0
	P_3	2 1 1	2 2 2	0 1 1
	P_4	0 0 2	4 3 3	4 3 1

<P_1, P_3, P_0, P_2, P_4> is a safe sequence.



Banker's Algorithm - Example (cont.)

- Suppose P_1 now requests (1, 0, 2).

Available		Allocation	Max	Need
A B C		A B C	A B C	A B C
2 3 0	P_0	0 1 0	7 5 3	7 4 3
	P_1	3 0 2	3 2 2	0 2 0
	P_2	3 0 2	9 0 2	6 0 0
	P_3	2 1 1	2 2 2	0 1 1
	P_4	0 0 2	4 3 3	4 3 1

<P_1, P_3, P_0, P_2, P_4> is a safe sequence also in this case.

What if P_4 then requests (3,3,0) ?
And if P_0 requests (0,2,0) ?

Yair Amir Fall 00 / Lecture 4 27

Program:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10],
safeSequence[10];
   int p, r, i, j, process, count;
   count = 0;
   printf("Enter the no of processes : ");
   scanf("%d", &p);
   for(i = 0; i< p; i++)
   completed[i] = 0;
   printf("\n\nEnter the no of resources : ");
   scanf("%d", &r);
   printf("\n\nEnter the Max Matrix for each process : ");
   for(i = 0; i < p; i++)
   {
    printf("\nFor process %d : ", i + 1);
    for(j = 0; j < r; j++)
    scanf("%d", &Max[i][j]);
}
printf("\n\nEnter the allocation for each process : ");
for(i = 0; i < p; i++)
{
    printf("\nFor process %d : ",i + 1);
    for(j = 0; j < r; j++)
    scanf("%d", &alloc[i][j]);
}

printf("\n\nEnter the Available Resources : ");
for(i = 0; i < r; i++)
scanf("%d", &avail[i]);
for(i = 0; i < p; i++)
for(j = 0; j < r; j++)
need[i][j] = Max[i][j] - alloc[i][j];
 do
    {
        printf("\n Max matrix:\tAllocation matrix:\n");
        for(i = 0; i < p; i++)
        {
            for( j = 0; j < r; j++)
            printf("%d ", Max[i][j]);
            printf("\t\t");
            for( j = 0; j < r; j++)
            printf("%d ", alloc[i][j]);
            printf("\n");
        }

        process = -1;
        for(i = 0; i < p; i++)
        {
            if(completed[i] == 0)//if not completed
            {
                process = i ;
                for(j = 0; j < r; j++)
                {
                    if(avail[j] < need[i][j])
                    {
```

```
                    process = -1;
                    break;
                }
            }
        }
        if(process != -1)
            break;
    }

    if(process != -1)
    {
        printf("\nProcess %d runs to completion!", process + 1);
        safeSequence[count] = process + 1;
        count++;
        for(j = 0; j < r; j++)
        {
            avail[j] += alloc[process][j];
            alloc[process][j] = 0;
            Max[process][j] = 0;
            completed[process] = 1;
        }
    }
}
while(count != p && process != -1);

if(count == p)
{
    printf("\nThe system is in a safe state!!\n");
    printf("Safe Sequence : < ");
    for( i = 0; i < p; i++)
    printf("%d ", safeSequence[i]);
    printf(">\n");
}
else
    printf("\nThe system is in an unsafe state!!");

}
```

Conclusion: Thus the Banker algorithm was executed and verified successfully.

Practical 8:

Aim: To implement Encryption Algorithm.

Theory:

Cryptography is a method of storing and transmitting data in a particular form so that only those for whom it is intended can read and process it. The term is most often associated with scrambling plaintext (ordinary text, sometimes referred to as cleartext) into ciphertext (a process called encryption), then back again (known as decryption).

Cryptography is a method of storing and transmitting data in a particular form so that only those for whom it is intended can read and process it.

Cryptography is closely related to the disciplines of cryptology and cryptanalysis. Cryptography includes techniques such as microdots, merging words with images, and other ways to hide information in storage or transit. However, in today's computer-centric world, cryptography is most often associated with scrambling plaintext (ordinary text, sometimes referred to as cleartext) into ciphertext (a process called encryption), then back again (known as decryption). Individuals who practice this field are known as cryptographers.

Modern cryptography concerns itself with the following four objectives:

1) **Confidentiality** (the information cannot be understood by anyone for whom it was unintended)

2) **Integrity** (the information cannot be altered in storage or transit between sender and intended receiver without the alteration being detected)

3) **Non-repudiation** (the creator/sender of the information cannot deny at a later stage his or her intentions in the creation or transmission of the information)

4) **Authentication** (the sender and receiver can confirm each other?s identity and the origin/destination of the information)

Encryption:

In cryptography, **encryption** is the process of encoding messages or information in such a way that only authorized parties can read it. Encryption does not of itself prevent interception, but denies the message content to the interceptor. In an encryption scheme, the message or information, referred to as plaintext, is encrypted using an encryption algorithm, generating ciphertext that can only be read if decrypted. For technical reasons, an encryption scheme usually uses a pseudo-random encryption_key generated by an algorithm. It is in principle possible to decrypt the message without possessing the key, but, for a well-designed encryption scheme, large computational resources and skill are required. An authorized recipient can easily decrypt the message with the key provided by the originator to recipients, but not to unauthorised interceptors.

Kinds of encryption:

Symmetric key encryption:

In symmetric-key schemes, the encryption and decryption keys are the same. Thus communicating parties must have the same key before they can achieve secret communication.

Public key encryption:

In public-key encryption schemes, the encryption key is published for anyone to use and encrypt messages. However, only the receiving party has access to the decryption key that enables messages to be read. Public-key encryption was first described in a secret document in 1973;before then all encryption schemes were symmetric-key (also called private-key).A publicly available public key encryption application called Pretty Good Privacy (PGP) was written in 1991 by Phil Zimmermann, and distributed free of charge with source code; it was purchased by Symantec in 2010 and is regularly updated.

Program:

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{
  int key,i;
  char data[30];
 clrscr();
 printf("\n enter the plain text");
 gets(data);
 printf("\n enter the key value");
 scanf("%d",&key);
{
   for(i=0;i<strlen(data);i++)
{
if(data[i]==' ')
{
}
else
{
if(data[i]>='x')
{
data[i]=data[i]-26;
}
data[i]=data[i]+key;
}
}
}
```

```
printf("\n your cipher text is %s",data);

getch();

}
```

Conclusion: Thus the Encryption algorithm was executed and verified successfully.

Practical 9:

Aim: To implement Dining philosophers Problem.

Theory: There is a dining room containing a circular table with five chairs. At each chair is a plate, and between each plate is a single chopstick. In the middle of the table is a bowl of spaghetti. Near the room are five philosophers who spend most of their time thinking, but who occasionally get hungry and need to eat so they can think some more. In order to eat, a philosopher must sit at the table, pick up the two chopsticks to the left and right of a plate, then serve and eat the spaghetti on the plate.

Thus, each philosopher is represented by the following pseudocode:

```
process P[i]
while true do
  { THINK;
   PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
   EAT;
   PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
  }
```

A philosopher may THINK indefinately. Every philosopher who EATs will eventually finish. Philosophers may PICKUP and PUTDOWN their chopsticks in either order, or nondeterministically, but these are atomic actions, and, of course, two philosophers cannot use a single CHOPSTICK at the same time.

Problems:

The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. To see that a proper solution to this problem isn't obvious, consider a proposal in which each philosopher is instructed to behave as follows:

think until the left fork is available; when it is, pick it up;

- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

This attempted solution fails because it allows the system to reach a deadlock state, in which no progress is possible. This is the state in which each philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available. With the given instructions, this state can be reached, and when it is reached, the philosophers will eternally wait for each other to release a fork.

Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of livelock. If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

Mutual exclusion is the core idea of the problem; the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources. These issues are studied in the branch of Concurrent Programming. The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties exemplified by the dining philosophers problem arise far more often when multiple processes access sets of data that are being updated. Systems such as operating system kernels use thousands of locks and synchronizations that require strict adherence to methods and protocols if such problems as deadlock, starvation, or data corruption are to be avoided.

Program:

```
#include<stdio.h>

#include<conio.h>

#include<dos.h>

#include<dir.h>

char fn2[20];
```

```
main()
{
 int c;
clrscr();
 do
{
 printf("\n\t\tMain Menu\n-------------------------------\n");
 printf("1.Copy a File\n2.Move a File\n3.Exit\n");
 scanf("%d",&c);
 switch(c)
 {
 case 1:
   copy_file();
    break;
 case 2:
   move_file();
    break;
 case 3:
   exit(0);
 }
 }
 while(c<=3);
 getch();
 return 0;
}
copy_file()
{
 FILE *f1,*f2;
 char ch,s[10],fn1[20];
 int a;
 printf("\nAre u see the privious files(1/0)?");
 scanf("%d",&a);
 if(a==1)
 print_file();
 printf("Enter the source file name:");
 scanf("%s",&fn1);
 printf("Enter the Destination file name:");
 scanf("%s",&fn2);
 f1=fopen(fn1,"r");
 if(f1==NULL)
 printf("Can't open the file");
 else
{
  f2=fopen(fn2,"w");
 while((ch=getc(f1))!=EOF)
 putc(ch,f2);
 printf("One File Copied");
 fclose(f2);
}
 fclose(f1);
 return 0;
}
move_file()
{
FILE *f1,*f2;
char ch,s[10],fn1[20];
int a;
printf("\nAre u see the privious files(1/0)?");
scanf("%d",&a);
if(a==1)
print_file();
printf("Enter the source file name:");
scanf("%s",&fn1);
printf("Enter the Destination file name:");
scanf("%s",&fn2);
f1=fopen(fn1,"r");
```

```
if(f1==NULL)
printf("Can't open the file");
else
{
 f2=fopen(fn2,"w");
 while((ch=getc(f1))!=EOF)
 putc(ch,f2);
 printf("One File moved");
 fclose(f2);
 remove(fn1);
}
 fclose(f1);
 return 0;
 }
print_file()
 {
 struct ffblk ffblk;
 int d,p=0;
 char ch;
 d=findfirst("*.*",&ffblk,0);
 while(!d)
  {
```

```
  printf("%s\n",ffblk.ff_name);
  d=findnext(&ffblk);
  p=p+1;
  if(p>=20)
  {
   printf("Press any key to continue");
   getchar();
   p=0;
  }
 }
 return 0;
}
```

Conclusion: Thus the Dining philosophers program was executed and verified successfully.

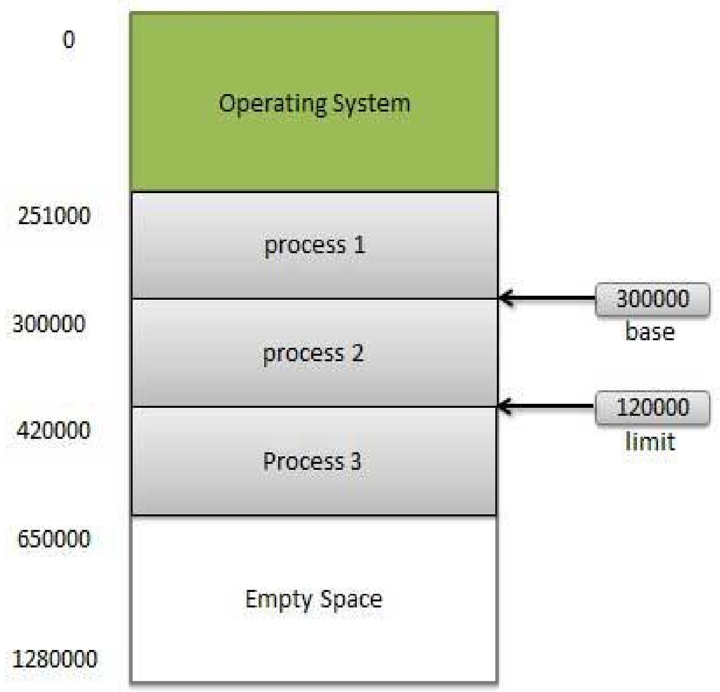
Practical 10:

Aim: To Study Memory Management.

Theory:

Memory management is the functionality of an operating system which handles or manages primary memory. Memory management keeps track of each and every memory location either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Memory management provides protection by using two registers, a base register and a limit register. The base register holds the smallest legal physical memory address and the limit register specifies the size of the range. For example, if the base register holds 300000 and the limit register is 1209000, then the program can legally access all addresses from 300000 through 411999.



Instructions and data to memory addresses can be done in following ways

- **Compile time** -- When it is known at compile time where the process will reside, compile time binding is used to generate the absolute code.
- **Load time** -- When it is not known at compile time where the process will reside in memory, then the compiler generates re-locatable code.
- **Execution time** -- If the process can be moved during its execution from one memory segment to another, then binding must be delayed to be done at run time

Dynamic Loading

In dynamic loading, a routine of a program is not loaded until it is called by the program. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed. Other routines methods or modules are loaded on request. Dynamic loading makes better memory space utilization and unused routines are never loaded.

Dynamic Linking

Linking is the process of collecting and combining various modules of code and data into a executable file that can be loaded into memory and executed. Operating system can link system level libraries to a program. When it combines the libraries at load time, the linking is called static linking and when this linking is done at the time of execution, it is called as dynamic linking. In static linking, libraries linked at compile time, so program code size becomes bigger whereas in dynamic linking libraries linked at execution time so program code size remains smaller.

Logical versus Physical Address Space

An address generated by the CPU is a logical address whereas address actually available on memory unit is a physical address. Logical address is also known a Virtual address.

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

The run-time mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory to a backing store , and then brought back into memory for continued execution.

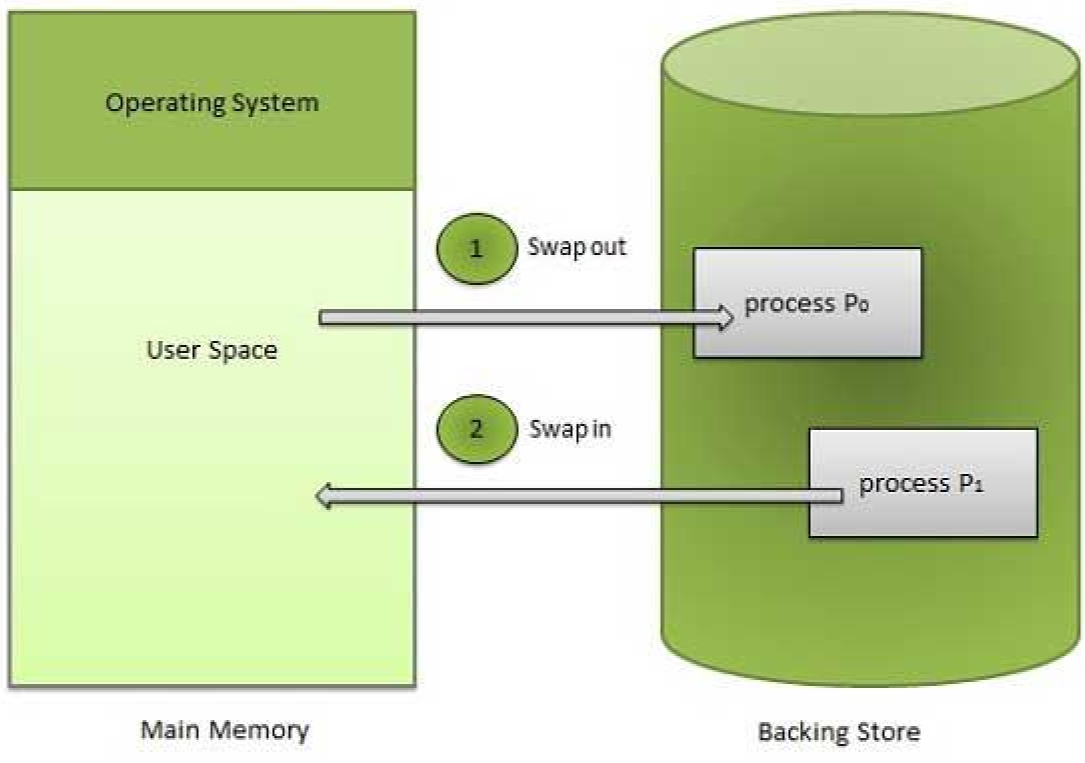
Backing store is a usually a hard disk drive or any other secondary storage which fast in access and large enough to accommodate copies of all memory images for all users. It must be capable of providing direct access to these memory images.

Major time consuming part of swapping is transfer time. Total transfer time is directly proportional to the amount of memory swapped. Let us assume that the user process is of size 100KB and the backing store is a standard hard disk with transfer rate of 1 MB per second. The actual transfer of the 100K process to or from memory will take

100KB / 1000KB per second

= 1/10 second

= 100 milliseconds.



Memory Allocation

Main memory usually has two partitions

- **Low Memory** -- Operating system resides in this memory.
- **High Memory** -- User processes then held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation	Description
1	**Single-partition allocation**	In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.

2	**Multiple-partition allocation**	In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes can not be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types

S.N.	Fragmentation	Description
1	**External fragmentation**	Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous so it can not be used.
2	**Internal fragmentation**	Memory block assigned to process is bigger. Some portion of memory is left unused as it can not be used by another process.

Paging

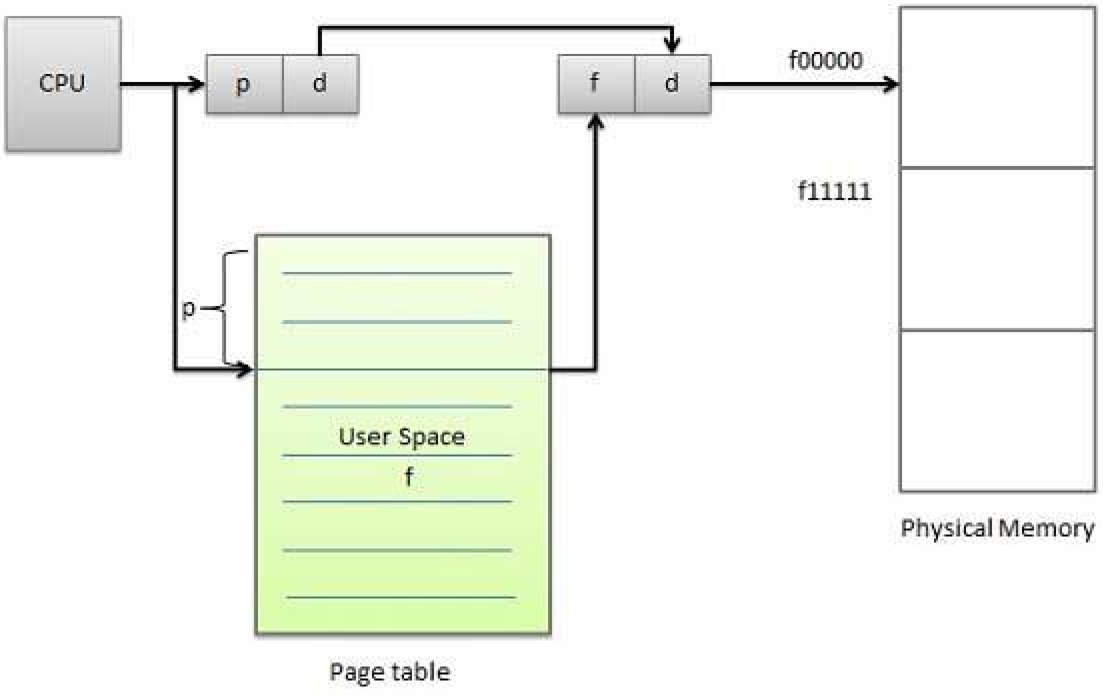
External fragmentation is avoided by using paging technique. Paging is a technique in which physical memory is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). When a process is to be executed, it's corresponding pages are loaded into any available memory frames.

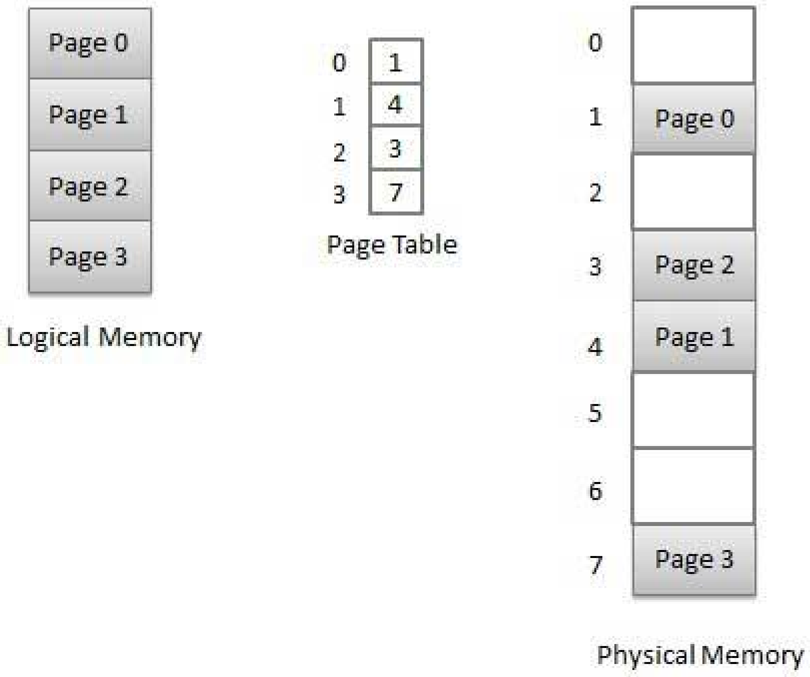
Logical address space of a process can be non-contiguous and a process is allocated physical memory whenever the free memory frame is available. Operating system keeps track of all free frames. Operating system needs n free frames to run a program of size n pages.

Address generated by CPU is divided into

- **Page number (p)** -- page number is used as an index into a page table which contains base address of each page in physical memory.

- **Page offset (d)** -- page offset is combined with base address to define the physical memory address.

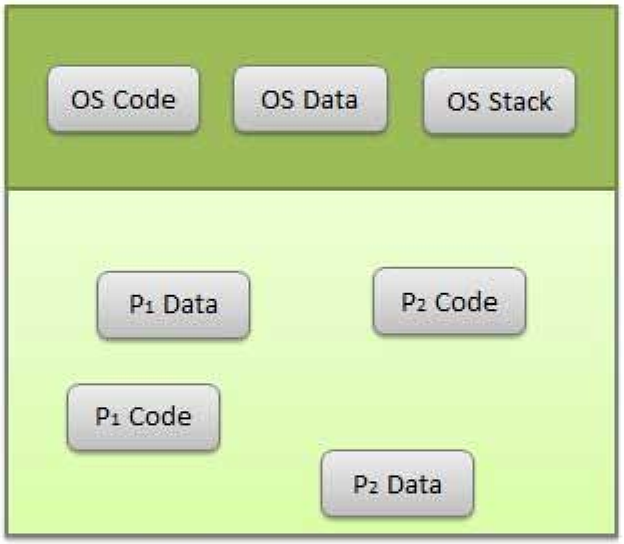




Segmentation

Segmentation is a technique to break memory into logical pieces where each piece represents a group of related information. For example ,data segments or code segment for each process, data segment for operating system and so on. Segmentation can be implemented using or without using paging.

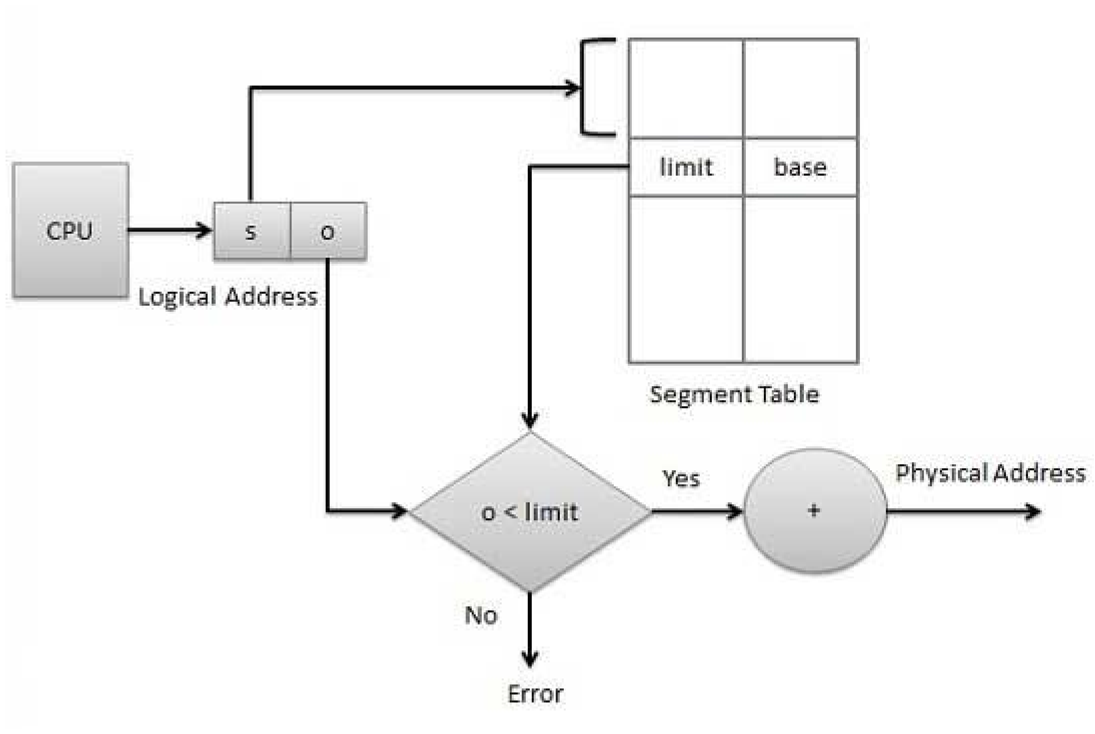
Unlike paging, segment are having varying sizes and thus eliminates internal fragmentation. External fragmentation still exists but to lesser extent.



Logical Address Space

Address generated by CPU is divided into

- **Segment number (s)** -- segment number is used as an index into a segment table which contains base address of each segment in physical memory and a limit of segment.
- **Segment offset (o)** -- segment offset is first checked against limit and then is combined with base address to define the physical memory address.



Conclusion: Thus we have Successfully Studied the Concept of Memory Management.