Vision of the Department

To be recognized for keeping innovation, research and excellence abreast of learning in the field of computer science & engineering to cater the global society.

Mission of the Department

M1: To provide an exceptional learning environment with academic excellence in the field of computer science and engineering.

M2: To facilitate the students for research and innovation in the field of software, hardware and computer applications and nurturing to cater the global society.

M3: To establish professional relationships with industrial and research organisations to enable the students to be updated of the recent technological advancements.

M4: To groom the learners for being the software professionals catering the needs of modern society with ethics, moral values and full of patriotism.

Program Educational Objectives (PEO's)

PEO1: The graduate will have the knowledge and skills of major domains of computer science and engineering in providing solution to real world problems most efficiently.

PEO2: The graduate will be able to create and use the modern tools and procedures followed in the software industry in the relevant domain.

PEO3: The graduate will be following the ethical practices of the software industry and contributing to the society as a responsible citizen.

PEO4: The graduate will have the innovative mindset of learning and implementing the latest developments and research outcomes in the computer hardware and software to keep pace with the fast changing socio economic world.

COURSE OUTCOMES

CO405.1 Determine the role of the operating system as a high level interface to the hardware.

CO405.2 Analyse FCFS, SSTF, SCAN and LOOK disk scheduling algorithm techniques

CO405.3 Implement FCFS, SJF, PRIORITY, RR CPU scheduling algorithm.

CO405.4 Summarize various memory management techniques.

CO405.5 Differentiate between Remote Procedure Call and Remote method invocation.

LIST OF EXPERIMENTS

1. Write a program to implement FCFS CPU scheduling algorithm.

2. Write a program to implement SJF CPU scheduling algorithm.

3. Write a program to implement Round Robin CPU scheduling algorithm.

4. Write a program to implement Priority CPU Scheduling algorithm.

5. Write a program to compare various CPU Scheduling Algorithms over different Scheduling Criteria.

6. Write a program to implement classical inter process communication problem (producer consumer).

7. Write a program to implement classical inter process communication problem (Reader Writers).

8. Write a program to implement classical inter process communication problem (Dining Philosophers).

9. Write a program to implement & Compare various page replacement algorithms.

10. Write a program to implement & Compare various Disk & Drum scheduling Algorithms.

EXPERIMENT -1

Aim: Write a program to implement FCFS CPU scheduling algorithm.

Source Code:

```
#include<stdio.h>
#include<conio.h>
int main ()
{
int bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
```

```
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
}
```

INPUT

Enter the number of processes -- 3

Enter Burst Time for Process 0 -- 24

Enter Burst Time for Process 1 -- 3

Enter Burst Time for Process 2 – 3

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUNDTIME
P0	24	0	24
P1	3	24	27
P2	3	27	30

Average Waiting Time-- 17.000000

Average Turnaround Time -- 27.000000

EXPERIMENT -2

Aim: Write a program to implement SJF CPU scheduling algorithm.

Source Code:

```
#include<stdio.h>
#include<conio.h>
main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg,
tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
```

```
temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=p[i];

p[i]=p[k];

p[k]=temp;

}

wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)

{

wt[i] = wt[i-1] +bt[i-1];

tat[i] = tat[i-1] +bt[i];

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n); getch();}
```

INPUT

Enter the number of processes -- 4

Enter Burst Time for Process 0 -- 6

Enter Burst Time for Process 1 -- 8

Enter Burst Time for Process 2 -- 7

Enter Burst Time for Process 3 -- 3

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P3	3	0	3
P0	6	3	9
P2	7	9	16
P1	8	16	24

Average Waiting Time -- 7.000000

Average Turnaround Time -- 13.000000

EXPERIMENT -3

Aim: Write a program to implement Round Robin CPU scheduling algorithm.

Source Code:

```
#include<stdio.h>

main()

{

int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;

float awt=0,att=0,temp=0;

clrscr();

printf("Enter the no of processes -- ");

scanf("%d",&n);

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

{

printf("\nEnter Burst Time for process %d -- ", i+1);

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

ct[i]=bu[i];

}

printf("\nEnter the size of time slice -- ");

scanf("%d",&t);

max=bu[0];

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

```
if(max<bu[i])

max=bu[i];

for(j=0;j<(max/t)+1;j++)

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

if(bu[i]!=0)

if(bu[i]<=t) {

tat[i]=temp+bu[i];

temp=temp+bu[i];

bu[i]=0;

}

else {

bu[i]=bu[i]-t;

temp=temp+t;

}

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

wa[i]=tat[i]-

ct[i]; att+=tat[i];

awt+=wa[i];}

printf("\nThe Average Turnaround time is -- %f",att/n);

printf("\nThe Average Waiting time is -- %f ",awt/n);

printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");

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

printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
```

getch();}

Page 8

INPUT:

Enter the no of processes – 3

Enter Burst Time for process 1 – 24

Enter Burst Time for process 2 -- 3

Enter Burst Time for process 3 – 3

Enter the size of time slice – 3

OUTPUT:

PROCESS	BURST TIME	WAITING TIME	TURNAROUNDTIME
1	24	6	30
2	3	4	7
3	3	7	10

The Average Turnaround time is – 15.666667

The Average Waiting time is - 5.666667

EXPERIMENT -4

Aim: Write a program to implement Priority CPU Scheduling algorithm.

Source Code:

```
#include<stdio.h>
main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp; float wtavg,
tatavg;
clrscr();
printf("Enter the number of processes --- ");
scanf("%d",&n);
for(i=0;i<n;i++){
p[i] = i;
printf("Enter the Burst Time & Priority of Process %d --- ",i); scanf("%d
%d",&bt[i], &pri[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(pri[i] > pri[k]){
temp=p[i];
p[i]=p[k];
    p[k]=temp;
    temp=bt[i];
    bt[i]=bt[k];
    bt[k]=temp;
    temp=pri[i];
    pri[i]=pri[k];
    pri[k]=temp;
    }
    wtavg = wt[0] = 0;
    tatavg = tat[0] = bt[0];
    for(i=1;i<n;i++)
    {
    wt[i] = wt[i-1] + bt[i-1];
    tat[i] = tat[i-1] + bt[i];
    wtavg = wtavg + wt[i];
    tatavg = tatavg + tat[i];
```

```
}

printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND
TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n); printf("\nAverage
Turnaround Time is --- %f",tatavg/n);
getch();}
```

INPUT
Enter the number of processes -- 5
Enter the Burst Time & Priority of Process 0 --- 10 3
Enter the Burst Time & Priority of Process 1 --- 1 1
Enter the Burst Time & Priority of Process 2 --- 2 4
Enter the Burst Time & Priority of Process 3 --- 1 5
Enter the Burst Time & Priority of Process 4 --- 5 2

OUTPUT

PROCESS	PRIORITY	BURST TIME	WAITING TIME	TURNAROUND TIME
1	1	1	0	1
4	2	5	1	6
0	3	10	6	16
2	4	2	16	18
3	5	1	18	19

Average Waiting Time is --- 8.200000
Average Turnaround Time is --- 12.000000

EXPERIMENT -5

Aim: Write a program to implement FIFO page replacement algorithm .

Source Code:

```
#include < stdio.h >
int main()
{
   int incomingStream[] = {4 , 1 , 2 , 4 , 5};
   int pageFaults = 0;
   int frames = 3;
   int m, n, s, pages;
   pages = sizeof(incomingStream)/sizeof(incomingStream[0]);
   printf(" Incoming \ t Frame 1 \ t Frame 2 \ t Frame 3 ");
   int temp[ frames ];
   for(m = 0; m < frames; m++)
   {
      temp[m] = -1;
   }
   for(m = 0; m < pages; m++)
   {
      s = 0;
      for(n = 0; n < frames; n++)
      {
         if(incomingStream[m] == temp[n])
         {
            s++;
```

```
            pageFaults--;
          }
        }
        pageFaults++;
        if((pageFaults <= frames) && (s == 0))
        {
          temp[m] = incomingStream[m];
        }
        else if(s == 0)
        {
          temp[(pageFaults - 1) % frames] = incomingStream[m];
        }
        printf("\n");
        printf("%d\t\t\t",incomingStream[m]);
        for(n = 0; n < frames; n++)
        {
          if(temp[n] != -1)
            printf(" %d\t\t\t", temp[n]);
          else
            printf(" - \t\t\t");
        }
    }
    printf("\nTotal Page Faults:\t%d\n", pageFaults);
    return 0;
}
```

Output

Incoming	frame1	frame2	frame3
4	4	-	-
1	4	1	-
2	4	1	2
4	4	1	2

5 5 1 2

Total page fault:- 4

EXPERIMENT -6

Aim: Write a program to implement classical inter process communication problem (producer consumer).

Source Code:

```
// C program for the above approach

#include <stdio.h>
#include <stdlib.h>

// Initialize a mutex to 1
int mutex = 1;

// Number of full slots as 0
int full = 0;

// Number of empty slots as size
// of buffer
int empty = 10, x = 0;

// Function to produce an item and
// add it to the buffer
void producer()
{
  // Decrease mutex value by 1
  --mutex;

  // Increase the number of full
  // slots by 1
  ++full;

  // Decrease the number of empty
  // slots by 1
  --empty;

  // Item produced
```

```
  x++;
  printf("\nProducer produces"
       "item %d",
       x);

  // Increase mutex value by 1
  ++mutex;
}

// Function to consume an item and
// remove it from buffer
void consumer()
{
  // Decrease mutex value by 1
  --mutex;

  // Decrease the number of full
  // slots by 1
  --full;

  // Increase the number of empty
  // slots by 1
  ++empty;
  printf("\nConsumer consumes "
       "item %d",
       x);
  x--;

  // Increase mutex value by 1
  ++mutex;
}

// Driver Code
int main()
{
  int n, i;
  printf("\n1. Press 1 for Producer"
       "\n2. Press 2 for Consumer"
       "\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'
// can give wrong value due to
// synchronization issues.

// 'critical' specifies that code is
// executed by only one thread at a
```

```
// time i.e., only one thread enters
   for (i = 1; i > 0; i++) {

        printf("\nEnter your choice:");
        scanf("%d", &n);

        // Switch Cases
        switch (n) {
        case 1:

                // If mutex is 1 and empty
                // is non-zero, then it is
                // possible to produce
                if ((mutex == 1)
                        && (empty != 0)) {
                        producer();
                }

                // Otherwise, print buffer
                // is full
                else {
                        printf("Buffer is full!");
                }
                break;

        case 2:

                // If mutex is 1 and full
                // is non-zero, then it is
                // possible to consume
                if ((mutex == 1)
                        && (full != 0)) {
                        consumer();
                }

                // Otherwise, print Buffer
                // is empty
                else {
                        printf("Buffer is empty!");
                }
                break;

        // Exit Condition
        case 3:
                exit(0);
                break;
```

```
        }
    }
}
```

EXPERIMENT -7

Aim: Write a program to implement classical inter process communication problem (Reader Writers).

Source Code:

Writer process:

Writer requests the entry to critical section. If allowed i.e. wait () gives a true value, it enters and performs the write. If not allowed, it keeps on waiting. It exits the critical section

```
do {
   // writer requests for critical section
   wait(wrt);
   // performs the write
   // leaves the critical section
   signal(wrt);
} while(true);
```

Reader process:

Reader requests the entry to critical section.

If allowed: It increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.

It then, signals mutex as any other reader is allowed to enter while others are already reading.

After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.

If not allowed, it keeps on waiting.

```
do {
   // Reader wants to enter the critical section
```

```
    wait(mutex);
    // The number of readers has now increased by 1
    readcnt++;
    // there is atleast one reader in the critical section
    // this ensure no writer can enter if there is even one reader
    // thus we give preference to readers here
    if (readcnt==1)
      wait(wrt);
    // other readers can enter while this current reader is inside
    // the critical section
    signal(mutex);
    // current reader performs reading here
    wait(mutex);   // a reader wants to leave
    readcnt--;
    // that is, no reader is left in the critical section,
    if (readcnt == 0)
       signal(wrt);         // writers can enter
    signal(mutex); // reader leaves
} while(true);
```

EXPERIMENT -8

Aim: Write a program to implement classical inter process communication problem (Dining Philosophers).

Source Code:

```
int tph, philname[20], status[20], howhung, hu[20], cho; main()

{

int i; clrscr();

printf("\n\nDINING PHILOSOPHER PROBLEM");

printf("\nEnter the total no. of philosophers: ");

scanf("%d",&tph);

for(i=0;i<tph;i++)

{

philname[i]=(i+1); status[i]=1;

}

printf("How many are hungry : ");

scanf("%d", &howhung);

if(howhung==tph)

{
```

```
printf(“\n All are hungry..\nDead lock stage will occur”);

printf(\n”Exiting\n”);

else{

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

printf(“Enterphilosopher%dposition:”,(i+1));

scanf(“%d”,&hu[i]);

status[hu[i]]=2;

}

do

{

printf("1.One can eat at a time\t2.Two can eat at a time

\t3.Exit\nEnter your choice:");

scanf("%d", &cho);

switch(cho)

{

case 1: one();

break;

case 2: two();

break;

case 3: exit(0);

default: printf("\nInvalid option..");

}

}while(1);
```

```
}
}
one()
{
int pos=0, x, i;
printf("\nAllow one philosopher to eat at any time\n");
for(i=0;i<howhung; i++, pos++)
{
printf("\nP %d is granted to eat", philname[hu[pos]]);
for(x=pos;x<howhung;x++)
printf("\nP %d is waiting", philname[hu[x]]);
}
}
two()
{
int i, j, s=0, t, r, x;
printf("\n Allow two philosophers to eat at same
time\n"); for(i=0;i<howhung;i++)
{
for(j=i+1;j<howhung;j++)
{
if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)
{
```

```
printf("\n\ncombination %d \n", (s+1));

t=hu[i];

r=hu[j]; s++;

printf("\nP %d and P %d are granted to eat", philname[hu[i]],

philname[hu[j]]);

for(x=0;x<howhung;x++)

{

if((hu[x]!=t)&&(hu[x]!=r))

printf("\nP %d is waiting", philname[hu[x]]);

}

}

}

}
```

INPUT:

DINING PHILOSOPHER PROBLEM

Enter the total no. of philosophers: 5

How many are hungry: 3

Enter philosopher 1 position: 2

Enter philosopher 2 position: 4

Enter philosopher 3 position: 5

OUTPUT:

1. One can eat at a time

2. Two can eat at a time

3. Exit Enter your choice: 1

Allow one philosopher to eat at any time

P 3 is granted to eat

P 3 is waiting

P 5 is waiting

P 0 is waiting

P 5 is granted to eat

P 5 is waiting

P 0 is waiting

P 0 is granted to eat

P 0 is waiting

EXPERIMENT -9

Aim: Write a program to implement & Compare various page replacement algorithms.

Source Code:

```
// C++ implementation of FIFO page replacement
// in Operating Systems.
#include<bits/stdc++.h>
using namespace std;
// Function to find page faults using FIFO
int pageFaults(int pages[], int n, int capacity)
{
        // To represent set of current pages. We use
        // an unordered_set so that we quickly check
        // if a page is present in set or not
        unordered_set<int> s;
        // To store the pages in FIFO manner
        queue<int> indexes;
        // Start from initial page
```

```
int page_faults = 0;
for (int i=0; i<n; i++)
{
        // Check if the set can hold more pages
        if (s.size() < capacity)
        {
                // Insert it into set if not present
                // already which represents page fault
                if (s.find(pages[i])==s.end())
                {
                        // Insert the current page into the set
                        s.insert(pages[i]);

                        // increment page fault
                        page_faults++;

                        // Push the current page into the queue
                        indexes.push(pages[i]);
                }
        }

        // If the set is full then need to perform FIFO
        // i.e. remove the first page of the queue from
        // set and queue both and insert the current page
        else
```

```
        {
                // Check if current page is not already
// present in the set
if (s.find(pages[i]) == s.end())
        {
// Store the first page in the
// queue to be used to find and
// erase the page from the set
int val = indexes.front();
// Pop the first page from the queue
indexes.pop();
// Remove the indexes page from the set
        s.erase(val);
// insert the current page in the set
s.insert(pages[i]);

        // push the current page into
                        // the queue
                        indexes.push(pages[i]);
                        // Increment page faults
                        page_faults++;
                }
        }
}
return page_faults;
```

```
}

// Driver code

int main()

{

        int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,

                                        2, 3, 0, 3, 2};

        int n = sizeof(pages)/sizeof(pages[0]);

        int capacity = 4;

        cout << pageFaults(pages, n, capacity);

        return 0;

}
```

EXPERIMENT -10

Aim: Write a program to implement & Compare various Disk & Drum scheduling Algorithm.

Source Code:

```
#include<iostream.h>
int main()
{
int i,j,sum=0,n;
int d[20];
int disk; //loc of head
int temp,max;
int dloc; //loc of disk in array
clrscr();
printf("enter number of location\t");
scanf("%d",&n);
printf("enter position of head\t");
scanf("%d",&disk);
```

```
printf("enter elements of disk queue\n");

for(i=0;i

{

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

}

d[n]=disk;

n=n+1;

for(i=0;i

{

for(j=i;j

{

if(d[i]>d[j])

{

temp=d[i];

d[i]=d[j];

d[j]=temp;

}

}



}

max=d[n];

for(i=0;i

{

if(disk==d[i]) { dloc=i; break; }
```

```
}
for(i=dloc;i>=0;i--)
{
printf("%d -->",d[i]);
}
printf("0 -->");
for(i=dloc+1;i
{
printf("%d-->",d[i]);
}
sum=disk+max;
printf("\nmovement of total cylinders %d",sum);
getch();
return 0;
}
```