Ex 1 - Parent Child Communication using Pipe

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
    int p[2];
    int pid;
    char inbuf[10],outbuf[10];
    pipe(p); //To send message between parent and child //
    pid=fork(); // Fork call to create child process //
    if(pid) //// Code of Parent process
    {
          printf("In parent process\n");
          printf("type the data to be sent to child");
          scanf("%s",outbuf); // Writing a message into the pipe
          write (p[1],outbuf, sizeof(outbuf)); //p[1] indicates write
          sleep(2); // To allow the child to run
          printf("after sleep in parent process\n");
    }
    else // Coding of child process //
    {
          sleep(2);
          printf("In child process\n");
          read(p[0],inbuf,10); // Read the message written by parent
          printf("the data received by the child is %s\n",inbuf);
    }
   return 0;
}
```

Ex 2A - Interprocess Communication using Shared Memory

SERVER

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<unistd.h>
#include<string.h>
#define SHMSZ 50
void main()
{
   char c;
   int shmid;
   key_t key;
   char*shm,*s;
   key=5678; // A random number used as key
   shmid=shmget(key,SHMSZ,IPC_CREAT|0666); // Create shared
//memory
   shm=(char*)shmat(shmid,NULL,0); //Attach shared memory
   s=shm; // Temporary pointer to avoid moving shm from base address of
//shared memory
   printf("Enter the message you want to send: ");
   scanf("%s", s); // Message copied into Shared memory directly through
//spointer
   while(*shm!='*') // Sender waits until received acknowledge it has read
//by appending * into shared memory
        sleep(1);
}
```

CLIENT

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
#define SHMSZ 50
void main()
{
    int shmid;
    key_t key;
    char *shm, *s;
    key = 5678;
    shmid = shmget(key, SHMSZ, 0666);
    shm = (char*)shmat(shmid, NULL, 0);
    for (s = shm; *s != '\0'; s++)
    putchar(*s);
         *shm = '*';
}
```

Ex 2B - Interprocess Communication using Message Queue

SENDER

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
// structure for message queue
struct mesg_buffer {
  long mesg_type;
  char mesg_text[100];
} message;
int main()
{
  key_t key;
  int msgid;
  // ftok to generate unique key
  key = ftok("progfile", 65);
```

```
  // msgget creates a message queue
  // and returns identifier
  msgid = msgget(key, 0666 | IPC_CREAT);
  //message.mesg_type = 1;
  printf("Writing Data : ");
  printf("\nEnter the message:");
  scanf("%s",message.mesg_text);
  do
  {
        printf("\nEnter the type for message:");
        scanf("%ld",&message.mesg_type);
        // msgsnd to send message
        msgsnd(msgid, &message, sizeof(message), 0);
        // display the message
        //printf("Data send is : %s \n", message.mesg_text);
        printf("\nEnter the message:");
        scanf("%s",message.mesg_text);
  }while(strcmp(message.mesg_text,"end")!=0);
 return 0;
}
```

RECEIVER

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
// structure for message queue
struct mesg_buffer {
  long mesg_type;
  char mesg_text[100];
} message;
int main()
{
  key_t key;
```

```
  int msgid,type;
  char choice[10];
  key = ftok("progfile", 65);
  msgid = msgget(key, 0666 | IPC_CREAT);
  printf("Read Data : ");
  do{
        printf("\nEnter the type of the message: ");
        scanf("%d",&type);
        msgrcv(msgid, &message, sizeof(message),type, 0);
        printf("\nMessage is : %s \n", message.mesg_text);
        printf("Do you want to continue: ");
        scanf("%s",choice);
        }while(strcmp(choice,"no")!=0);
  msgctl(msgid, IPC_RMID, NULL);
  return 0;
}
```

Ex 3 - CPU Scheduling Algorithms

NON PREEMPTIVE :

FCFS(First come First serve) :

```
#include <stdio.h>

struct process
{
    int at;
    int st;
    int status;
    int ft;
}ready_list[10];
int n;
int dispatcher(int time)
{
    int i,lat = time,index=-1;
```

```
        for(i=0;i<n;i++)
    {
        if(ready_list[i].status != 1){
            if(ready_list[i].at <= lat)
                {
                    lat = ready_list[i].at;
                    index=i;
                }
        }


    }
    return index;
}
int main()
{
    int i,cur_time,pid;
    printf("Enter number of processes:");
    scanf("%d",&n);
    // Collect process details -
    for(i=0;i<n;i++)
    {
        printf("Process %d\n",i+1);
        printf("***********\n");
        printf("Enter Arrival Time:");
        scanf("%d",&ready_list[i].at);
        printf("Enter Service Time:");
        scanf("%d",&ready_list[i].st);
        ready_list[i].status=0;
    }
    i=0; cur_time=0;
    while(i < n)
    {
        pid=dispatcher(cur_time);
        while(pid==-1){
            cur_time++;
            pid = dispatcher(cur_time);
```

```
        }
        ready_list[pid].ft=cur_time + ready_list[pid].st;
        ready_list[pid].status=1;
        cur_time+= ready_list[pid].st;
        i++;
    }
    printf("Process\t Arrival Time\t Burst Time\tFinish Time \t TT \t\t WT\n");
    printf("*******\t ************\t ************\t ***********\t********\t********\n");
    for(i=0;i<n;i++)
    {

printf("%d\t\t%d\t\t%d\t\t%d\t%d\t\t%d\n",i+1,ready_list[i].at,ready_list[i].st,ready_l
ist[i].ft, (ready_list[i].ft-ready_list[i].at),(ready_list[i].ft-ready_list[i].at)
-ready_list[i].st);
    }
}
```

SJF(Shortest Job First) :

```
#include <stdio.h>
#include<limits.h>
struct process
{
    int at;
    int st;
    int status;
    int ft;
}ready_list[10];
int n;
int dispatcher(int time)
{
    int i,bt = INT_MAX,index=-1;
    for(i=0;i<n;i++)
    {
        if(ready_list[i].status != 1)
            if(ready_list[i].at <= time)
                if(ready_list[i].st<=bt)
```

```
                        {
                                bt = ready_list[i].st;
                                index=i;
                        }
    }
    return index;
}
int main()
{
    int i,cur_time,pid;
    printf("Enter number of processes:");
    scanf("%d",&n);
    // Collect process details -
    for(i=0;i<n;i++)
    {
        printf("Process %d\n",i+1);
        printf("***********\n");
        printf("Enter Arrival Time:");
        scanf("%d",&ready_list[i].at);
        printf("Enter Service Time:");
        scanf("%d",&ready_list[i].st);
        ready_list[i].status=0;
    }
    i=0; cur_time=0;
    while(i < n)
    {
        pid=dispatcher(cur_time);
        while(pid==-1){
                cur_time++;
                pid = dispatcher(cur_time);
        }
        ready_list[pid].ft=cur_time + ready_list[pid].st;
        ready_list[pid].status=1;
        cur_time+= ready_list[pid].st;
        i++;
    }
    printf("Process\t Arrival Time\t Burst Time\tFinish Time \t TT \t\t WT\n");
```

```
    printf("*******\t ************\t ************\t ***********\t********\t********\n");
    for(i=0;i<n;i++)
    {

printf("%d\t\t%d\t\t%d\t\t%d\t%d\t\t%d\n",i+1,ready_list[i].at,ready_list[i].st,ready_l
ist[i].ft, (ready_list[i].ft-ready_list[i].at),(ready_list[i].ft-ready_list[i].at)
-ready_list[i].st);
    }
}
```

PREEMPTIVE :

SRTF(Shortest remaining time first) :

```
#include <stdio.h>

struct process
{
   int at;
   int st;
   int status;
   int ft;
}ready_list[10];
int n;
int dispatcher(int time)
{
   int i,s_bt=9999,index=-1;
   for(i=0;i<n;i++)
   {
     if(ready_list[i].status != 1)
        if(ready_list[i].at <= time)
           if(ready_list[i].st < s_bt)
           {
              s_bt = ready_list[i].st;
              index=i;
           }
    }
return index;
}

int main()
{
```

```
  int i,cur_time,pid;
  int rem_procs=0;
  printf("Enter number of processes:");
  scanf("%d",&n);
  // Collect process details -
  for(i=0;i<n;i++)
  {
    printf("Process %d\n",i+1);
    printf("***********\n");
    printf("Enter Arrival Time:");
    scanf("%d",&ready_list[i].at);
    printf("Enter Burst Time:");
    scanf("%d",&ready_list[i].st);
    ready_list[i].status=0;
  }
  int bd[n];
  for(i=0;i<n;i++){
     bd[i] = ready_list[i].st;
  }
  cur_time=0;
  while(rem_procs < n)
  {
    pid=dispatcher(cur_time);
    ready_list[pid].ft=cur_time + 1;
    cur_time = cur_time + 1;
    ready_list[pid].st-=1;
    if(ready_list[pid].st==0)
    {
       ready_list[pid].status=1;
       rem_procs+=1;
    }
   }
 printf("Process\t Arrival Time\t Burst Time\t Finish Time");
 printf("*******\t **********\t **********\t *********\n");
 for(i=0;i<n;i++)
 {
   printf("%d\t\t%d\t\t%d\t\t\t\t%d\n",i,ready_list[i].at,bd[i],ready_list[i].ft);
 }
}
```

RR(Round robin) :

```
#include<stdio.h>
struct process
{
  int at;
  int st;
  int ft;
  int status;
}ready[10];
int n,t,com=0;
int Dispatch(int ct)
{
  int i,index=-1,high_at=0,high_status=0; //high_status - 0 - Not yet execsted once
, 1- already executed atleast once
  int m;
  m=n;
  for(i=0;i<n;i++)
  {
   if(ready[i].at> high_at)
   {
         high_at=ready[i].at;
   }
  }
  for(i=0;i<n;i++)
  {
   if(ready[i].status > high_status)
   {
         high_status=ready[i].status;
   }
  }
  for(i=0;i<n;i++)
  {
   if(ready[i].status!=2) //Status = 2 means process already completed
   {
         if(ready[i].at<=ct)
         {
         if(ready[i].at<high_at)
         {
```

```
        index=i;
        high_at=ready[i].at;
        }
        if(ready[i].at==high_at)
        {
        if(ready[i].status<high_status)
        {
        index=i;
        high_status=ready[i].status;
        }
        else if(ready[i].status==high_status)
        {
        if(i<m)
        {
        index=i;
        m=i;
        }
        }
        }
        }
   }
 }
 return index;
}
int main()
{
  int i;
  printf("Enter number of processes:");
  scanf("%d",&n);
  printf("Enter the time slice:");
  scanf("%d",&t);
  for(i=0;i<n;i++)
  {
        printf("Process:%d\n",i+1);
        printf("**********\n");
        printf("Enter the arrival time:");
        scanf("%d",&ready[i].at);
```

```
        printf("Enter the service time:");
        scanf("%d",&ready[i].st);
        ready[i].status=0;
  }
  i=0;
  int at[10],st[10];
  for(i=0;i<n;i++)
  {
        at[i]=ready[i].at;
        st[i]=ready[i].st;
  }
  int pid,cur_time=0;
  while(com<n)
  {
   pid=Dispatch(cur_time);
   if(ready[pid].st<=t)
   {
        cur_time+=ready[pid].st;
        ready[pid].ft=cur_time;
        ready[pid].status=2;
        com++;
   }
   else
   {
        cur_time+=t;
        ready[pid].at=cur_time;
        ready[pid].st=ready[pid].st-t;
        ready[pid].status=1;
   }
  }
 printf("process-id\t arrival time\t service time\t finish time\t turnaround time
\twaitingtime\n");

printf("**********************************************************************************
***************\n");
 for(i=0;i<n;i++)
 {
```

```
   printf("%d\t\t %d\t\t\t %d\t\t %d\t\t %d\t\t\t\t
%d\n",i+1,at[i],st[i],ready[i].ft,ready[i].ft-at[i],ready[i].ft-at[i]-st[i]);
 }
}
```

Ex 4 - Multi processor scheduling

```
#include <stdio.h>
struct process
{
    int at;
    int st;
    int cpu;
    int status;
    int ft;
}ready_list[10];
int n;
int main()
{
    int i,j,pid,h;
    printf("Enter number of processes:");
    scanf("%d",&n);
    printf("Enter no of cpu");
    scanf("%d",&h);
    // Collect process details -
    for(i=0;i<n;i++)
    {
        printf("Process %d\n",i+1);
        printf("*****\n");
        printf("Enter Arrival Time:");
        scanf("%d",&ready_list[i].at);
        printf("Enter Service Time:");
        scanf("%d",&ready_list[i].st);
        ready_list[i].status=0;
```

```
    }
    i=0;
    int cur_time[h];
    int m=0;
    for(m=0;m<h;m++){
        cur_time[m]=0;
    }
    while(i < n)
    {
        for(j=0;j<h;j++)
        {
        pid=dispatcher(cur_time[j]);
        while(pid==-1){
            cur_time[j]++;
            pid=dispatcher(cur_time[j]);
        }
        if(pid!=-1){
        ready_list[pid].ft=cur_time[j]+ ready_list[pid].st;
        ready_list[pid].status=1;
        ready_list[pid].cpu=j+1;
        cur_time[j]+= ready_list[pid].st;
        i++;
        }
        }
    }
    printf("Process\t Arrival Time\t Burst Time\t    Finish Time \t CPU \t TT \t
WT\n");
    printf("**\t ****\t ****\t ****\n");
    for(i=0;i<n;i++)
    {

printf("%d\t\t%d\t\t%d\t\t\t%d\t%d\t%d\t%d\n",i,ready_list[i].at,ready_list[i].st,
ready_list[i].ft,ready_list[i].cpu,
(ready_list[i].ft-ready_list[i].at),(ready_list[i].ft-ready_list[i].at)
-ready_list[i].st);
```

```
    }
}
int dispatcher(int time)
{
    int i,index=-1;
    for(i=0;i<n;i++)
    {
        if(ready_list[i].status != 1)
            if(ready_list[i].at <= time)
                {
                    index=i;
                    return index;
                }
    }
    return index;
}
```

Ex 5a - Producer Consumer Problem(Single producer Single consumer)
[Compilation syntax : cc -pthread filename.c (or) gcc filename.c]

```
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

int front = 0;
int rear = -1;
int array[5];
```

```
sem_t mutex;
sem_t emptyCount;
sem_t fullCount;

int produce_item()
{
        static int a=100;
        return a++;
}
void insert_item(int item)
{
        rear = rear+1;
        rear = rear % 5;
        array[rear] = item;
}
int remove_item()
{
        int item= array[front];
        front = front+1;
        front = front%5;
        return item;
}
void * produce()
{
   int item;
   while(1)
   {
        item=produce_item();
        sem_wait(&emptyCount); // to see whether empty spaces available
        sem_wait(&mutex); // to ensure that consumer is not using the buffer
        printf("\nProducer entering the critical section");
        insert_item(item);
        printf("\nProducer inserting an item %d at %d",item, rear);
        sem_post(&mutex); // release mutex to let consumer to access buf
        sem_post(&fullCount); // to let consumer take the new data
```

```
    }
}

void *consumer()
{
        int item;
        while(1)
        {
          sleep(5);
          sem_wait(&fullCount); //to see whether there is data in buffer
          sem_wait(&mutex); // to ensure that producer is not using the buffer
          printf("\nConsumer entering the critical regiion");
          item=remove_item();
          printf("\nConsumer consumed item %d",item);
          sem_post(&mutex); // release mutual exclusion
          printf("\nConsumer leaving the critical region");
          sem_post(&emptyCount); // Increment no of empty slots and
//unblock producer if it was blocked
         }
}

int main()
{
        pthread_t p_tid;
        pthread_t c_tid;
        sem_init(&mutex,0,1);
        sem_init(&emptyCount,0,5);
        sem_init(&fullCount,0,0);
        pthread_create(&p_tid,NULL,produce,0);
        pthread_create(&c_tid,NULL,consumer,0);
        pthread_join(p_tid,NULL);
        pthread_join(c_tid,NULL);
        return 0;
}
```

Ex 5b - Reader Writer Problem

```
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

sem_t mutex;
sem_t rw_mutex;

int readcount=0;
int ticket = 0;

void book()
{    ticket++;
}
void cancel()
{    ticket--;
}
void *reader()
{    while(1)
   {    sem_wait(&mutex);
        readcount++;
        if(readcount==1)
                sem_wait(&rw_mutex);
        sem_post(&mutex);
        printf("\nTicket value = %d",ticket);
        sem_wait(&mutex);
        readcount--;
        if(readcount==0)
                sem_post(&rw_mutex);
        sem_post(&mutex);
        sleep(3);
```

```
    }
}
void *writer()
{    char op;
    while(1)
    {    sleep(3);
         sem_wait(&rw_mutex);
         printf("\nEnter b to book, c to cancel : ");
         scanf(" %c",&op);
         if(op=='b')
                 book();
         if(op=='c')
                 cancel();
         sem_post(&rw_mutex);
    }
}

int main()
{   pthread_t r_tid;
    pthread_t w_tid;
    sem_init(&mutex, 0 ,1);
    sem_init(&rw_mutex, 0, 1);
    pthread_create(&r_tid, NULL, reader, 0);
    pthread_create(&w_tid, NULL, writer, 0);
    pthread_join(r_tid, NULL);
    pthread_join(w_tid, NULL);
    return 0;
}
```

Ex 5c - Implementing Dining Philosophers Problem

```
#include <stdio.h>
#include <semaphore.h>
#include <unistd.h>
```

```
#include <pthread.h>

#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)% N
#define THINKING 0
#define HUNGRY 1
#define EATING 2

int i=1;
int state[N];
sem_t mutex;
sem_t s[N];

void * test(int i)
{
if(state[i]== HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING)
{
state[i]= EATING;
sem_post(&s[i]);
}
}

void * take_forks(int i)
{
sem_wait(&mutex);
state[i] = HUNGRY;
printf("\n philosopher %d is in hungry state",i);
test(i);
sem_post(&mutex);
sem_wait(&s[i]);

}
```

```
void * put_forks(int i)
{
sem_wait(&mutex);
state[i] = THINKING;
test(LEFT);
test(RIGHT);
sem_post(&mutex);
}

void * philosopher(void *i)
{
int *p = (int *) i;
while(1)
{
printf("\n philosopher %d is thinking",*p);
sleep(5);
take_forks(*p);
printf("\n philosopher %d is eating",*p);
sleep(5);
put_forks(*p);
}
}

int main()
{
pthread_attr_t *attr= NULL;
pthread_t p_tid1,p_tid2,p_tid3,p_tid4,p_tid5;
sem_init(&mutex,0,1);
int p[5]={0,1,2,3,4};
pthread_create(&p_tid1,attr,philosopher,(void *) &p[0]);
pthread_create(&p_tid2,attr,philosopher,(void *) &p[1]);
pthread_create(&p_tid3,attr,philosopher,(void *) &p[2]);
pthread_create(&p_tid4,attr,philosopher,(void *) &p[3]);
pthread_create(&p_tid5,attr,philosopher,(void *) &p[4]);
pthread_join(p_tid1,NULL);
```

```
pthread_join(p_tid2,NULL);
pthread_join(p_tid3,NULL);
pthread_join(p_tid4,NULL);
pthread_join(p_tid5,NULL);
return 0;
}
```

Ex 6 - Implementing Banker's Algorithm

TYPE 1(Algorithm which checks for safety of system alone):-

```
#include <stdio.h>

int need[5][5], max[5][5], allocation[5][5],avl[5];
int allocated_resources[5] = {0,0,0,0,0};
int maxres[5], completed[5], safe=0;
int count = 0, i, j, exec, r, p;

int main()
{
  printf("\nEnter the number of processes: ");
  scanf("%d",&p);
  for(i=0;i<p;i++)
  {
       completed[i]=0;
       count++;
  }
  printf("\nEnter the number of resources: ");
  scanf("%d",&r);
  printf("\nEnter the available (AVL) number of instances of resource");
  for(i=0;i<r;i++)
  {
       printf("Resource:%d: ",i);
       scanf("%d",&avl[i]);
```

```
}

printf("\nEnter maximum resource( Max) demand of each process:\n");
for(i=0;i<p;i++)
{
    printf("Process %d :\n",i);
    for(j=0;j<r;j++)
    {
    scanf("%d",&max[i][j]);
    }
}

printf("\nEnter already allocated resource table (Allocation) ):\n");
for(i=0;i<p;i++)
{
    printf("Process %d:\n",i);
    for(j=0;j<r;j++)
    {
    scanf("%d",&allocation[i][j]);
    }
}
int x=0;
for(i=0;i<r;i++)
{
    for(j=0;j<p;j++)
    {
    allocated_resources[i]+=allocation[j][i];
    }
}
for(i=0;i<r;i++)
{
    maxres[i]=avl[i]+allocated_resources[i];
}
printf("\nMaximum resources:");
for(i=0;i<r;i++)
```

```
  {
        printf("\t%d",maxres[i]);
  }
  printf("\n");
  for(i=0;i<p;i++)
        for(j=0;j<r;j++)
        need[i][j]=max[i][j]-allocation[i][j];
//Main procedure goes below to check for unsafe state.
  while(count!=0)
  {
        safe=0;

        for(i=0;i<p;i++)
        {
        if(!completed[i])
        {
        exec=1;
        for(j=0;j<r;j++)
        {
        if(need[i][j] > avl[j]){
                exec=0;
                break;

        }
        }
        if(exec) // Process i can be completed with available resources
        {
        printf("P%d\t",i);
        completed[i]=1; // Process i executed with available resources
        count--;
        safe=1;

        for(j=0;j<r;j++) {
                avl[j]+=allocation[i][j];
        }
```

```
        break;

        }
        }
        }
        if(!safe)
        {
        printf("\nThe system is in unsafe state.\n");
        break;
        }
   }
   if(safe)
    printf("\nThe system is in safe state");
}
```

TYPE 2(Algorithm which checks for safety of system as well as checks whether a request can be granted or not):-

```
#include<stdio.h>
struct process{
    int max[100],alloc[100],need[100];
    int status;
}p[100];
int nr,np,k;
int avai[100]={0};
int work[100]={0};
int req[100]={0};
int safeSequence[100]={-1};
int si=0;
void display(){
    printf("Process\tMaximum\tAllocated\tNeed\n");
    int i,j;
    for(i=0;i<np;i++){
        printf("P%d\t",i+1);
        for(j=0;j<nr;j++){
```

```
            printf("%d ",p[i].max[j]);
        }
        printf("\t");
        for(j=0;j<nr;j++){
            printf("%d ",p[i].alloc[j]);
        }
        printf("\t\t");
        for(j=0;j<nr;j++){
            printf("%d ",p[i].need[j]);
        }
        printf("\n");
    }
    printf("\nAvailable:\n");
    for(j=0;j<nr;j++){
        printf("%d\t",avai[j]);
    }
    printf("\n");
}
int resoureRequest(){
    int j;
    for(j=0;j<nr;j++){
        if (req[j]>p[k].need[j]){
            return 0;
        }
    }
    for(j=0;j<nr;j++){
        if (req[j]>avai[j]){
            return 0;
        }
    }
    for(j=0;j<nr;j++){
        avai[j]-=req[j];
        work[j]-=req[j];
        p[k].alloc[j]+=req[j];
        p[k].need[j]-=req[j];
```

```
    }
    return 1;
}
int safety(){
    int i,j,flag,flag2,cp=0;
    while(cp<np){
        flag=0;
        for (i=0;i<np;i++){
            flag2=1;
           if(p[i].status==0){
               for(j=0;j<nr;j++){
                   if (p[i].need[j]>work[j]){
                       flag2=0;
                      break;
                   }
               }
               if(flag2==1){
                   cp++;
                   safeSequence[si]=i+1;
                   si++;
                   p[i].status=1;
                   flag=1;
                   for(j=0;j<nr;j++){
                       printf("%d ",p[i].need[j]);
                   }

                   for(j=0;j<nr;j++){
                       work[j]+=p[i].alloc[j];
                       printf("%d ",work[j]);
                   }
                    printf("\n");
               }
           }
        }
        if (flag==0)
```

```
            return 0;
        }
        return 1;
    }
    int main(){
        printf("Enter number of processes:");
        scanf("%d",&np);
        printf("Enter number of resources:");
        scanf("%d",&nr);
        int i,j,op;
        for(i=0;i<np;i++){
            printf("For process P%d\n",i+1);
            for(j=0;j<nr;j++){
                printf("Enter maximum demand for resource %d:",j+1);
                scanf("%d",&p[i].max[j]);
                printf("Enter  allocated instances for resource %d:",j+1);
                scanf("%d",&p[i].alloc[j]);
                p[i].need[j]=p[i].max[j]-p[i].alloc[j];
                p[i].status=0;
            }
        }
        for(j=0;j<nr;j++){
                printf("Enter available instances for resource %d:",j+1);
                scanf("%d",&avai[j]);
                work[j]=avai[j];
        }
        printf("Are additinal resources are requested?(1.Yes/2.No)");
        scanf("%d",&op);
        if(op==1){
            printf("Process requesting:");
            scanf("%d",&k);
        for(j=0;j<nr;j++){
             printf("Enter additional demand for resource %d:",j+1);
             scanf("%d",&req[j]);
            }
```

```
        if (resoureRequest()==1)
            printf("Resources can be granted.\n");
        else
            printf("Resources can not be granted.\n");
    }
    display();
    if(safety()==1){
        printf("Processes are in safe state.\n");
        printf("Safe sequence:\n");
        for(i=0;i<np;i++)
            printf("%d ",safeSequence[i]);
    }
    else
        printf("Processes are in unsafe state.\n");
    return 0;
}
```

Ex 7 - Page Replacement Techniques

Optimal :-

```
#include <stdio.h>
#include <limits.h>
int n;
struct RAM{
    int pno;
}frame[10];
int q_ptr=0;
void init(){
    int i;
    for(i=0;i<n;i++)
        frame[i].pno=-1;
}
```

```
void display(){
    int i;
    printf("\nFrame(RAM)");
    for(i=0;i<n;i++)
        printf("\n%d",frame[i].pno);
}
int isinRAM(int p_no){
    int i,flag=0;
    for(i=0;i<n;i++){
        if(frame[i].pno==p_no){
            return i;
        }
    }
    return -1;
}
int main(){
    int pf=0;
    int np;
    printf("Enter the number of frames in memory:    ");
    scanf("%d",&n);
    printf("Enter the number of page requests:    ");
    scanf("%d",&np);
    init();
    int page_req[np];
    printf("\nEnter the Page number requested one by one:    ");
    for(int i=0;i<np;i++){
        scanf("%d",&page_req[i]);
    }
    display();
    int next_occ[n];
    for(int i=0;i<np;i++){
        if(page_req[i]==-1)
            break;
```

```
if(isinRAM(page_req[i])!=-1){
   printf("\nPage Exists...");
}
else{
   pf++;
   printf("\nPage Fault %d",pf);
   if(q_ptr<n){
      frame[q_ptr].pno=page_req[i];
      q_ptr++;
   }
   else{
      for(int i=0;i<n;i++){
         next_occ[i]=INT_MAX;
      }
      for(int j=0;j<n;j++){
         for(int k=i+1;k<np;k++){
            if(page_req[k]==frame[j].pno){
               next_occ[j]=k;
               break;
            }
         }
      }
      int max_occ=next_occ[0];
      int process_replace=0;
      for(int j=0;j<n;j++){
         if(next_occ[j]>max_occ){
            max_occ=next_occ[j];
            process_replace=j;
         }
      }
      frame[process_replace].pno=page_req[i];
   }
}
```

```
        display();
    }
    return 0;
}
```

LRU :-

```
#include <stdio.h>
#include <limits.h>
int n;
struct RAM{
        int pno;
}frame[10];
int q_ptr=0;
void init(){
        int i;
        for(i=0;i<n;i++)
        frame[i].pno=-1;
}
void display(){
        int i;
        printf("\nFrame(RAM)");
        for(i=0;i<n;i++)
        printf("\n%d",frame[i].pno);
}
int isinRAM(int p_no){
        int i,flag=0;
        for(i=0;i<n;i++){
        if(frame[i].pno==p_no){
        return i;
        }
        }
        return -1;
```

```
}
int main(){
        int pf=0;
        int np;
        printf("Enter the number of frames in memory: ");
        scanf("%d",&n);
        printf("Enter the number of page requests:       ");
        scanf("%d",&np);
        init();
        int page_req[np];
        printf("\nEnter the Page number requested one by one:      ");
        for(int i=0;i<np;i++){
        scanf("%d",&page_req[i]);
        }
        display();
        int prev_occ[n];
        for(int i=0;i<np;i++){
        if(page_req[i]==-1)
        break;
        if(isinRAM(page_req[i])!=-1){
        printf("\nPage Exists...");
        }
        else{
        pf++;
        printf("\nPage Fault %d",pf);
        if(q_ptr<n){
                frame[q_ptr].pno=page_req[i];
                q_ptr++;
        }
        else{
                for(int i=0;i<n;i++){
                prev_occ[i]=INT_MAX;
                }
```

```
            for(int j=0;j<n;j++){
            for(int k=i-1;k>=0;k--){
            if(page_req[k]==frame[j].pno){
                    prev_occ[j]=k;
                    break;
            }
            }
            }
            int min_occ=prev_occ[0];
            int process_replace=0;
            for(int j=0;j<n;j++){
            if(prev_occ[j]<min_occ){
            min_occ=prev_occ[j];
            process_replace=j;
            }
            }
            frame[process_replace].pno=page_req[i];
        }
        }
        display();
        }
        return 0;
}
```

Additional Excercises :-

Passing integer array from parent to child

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
```

```
void main(){
    int p[2];
    int n;
    pipe(p);
    int pid;
    int i;
    int inp[20], op[20];
    printf("Enter size of array : ");
    scanf("%d",&n);
    printf("Enter the elements : ");
    for(i=0;i<n;i++)
        scanf("%d",&inp[i]);
    pid = fork();
    if(pid){
        printf("In parent process\n");
        write(p[1],inp,sizeof(inp));
        sleep(3);
        printf("After sleep in parent process");
    }
    else{
        sleep(3);
        printf("In child process\n");
        read(p[0],op,sizeof(op));
        printf("The array sent by parent is ");
        int j;
        for(j=0;j<n;j++)
            printf("%d ",op[j]);
    }
}
```

Two way communication between parent and child

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

```
#include<sys/types.h>
#include<unistd.h>

void main(){
   int p1[2],p2[2];
   pipe(p1);
   pipe(p2);
   int pid;
   char Pin[15],Pout[15],Cin[15],Cout[15];
   int rtck = 0;
   pid = fork();
   if(pid){
l1 :
        if(rtck==0){
                printf("Enter message(from parent to child) : ");
                scanf("%s",Pin);
                write(p1[1],Pin,sizeof(Pin));
                sleep(3);
        }
        if(rtck==1){
                sleep(3);
                read(p2[0],Cout,sizeof(Cout));
                printf("Child said %s",Cout);
                rtck = -1;
        }
   }
   else{
        if(rtck==0){
                sleep(3);
                read(p1[0],Pout,sizeof(Pout));
                printf("Parent said %s",Pout);
                rtck = 1;
        }
        if(rtck==1){
                printf("Enter message(from child to parent) : ");
```

```
                scanf("%s",Cin);
                write(p2[1],Cin,sizeof(Cin));
                sleep(3);
                goto l1;
        }
    }
}
```

Implementing pipe using shared memory

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<unistd.h>
#include<stdlib.h>

#define SHMSZ 50

void main(){
    int pid, shmid;
    key_t key;
    key = 9000;
    char *shm, *s;
    pid = fork();
    if(pid){
        printf("In parent process :-\n");
        shmid = shmget(key,SHMSZ,IPC_CREAT|0666);
        shm = (char*)shmat(shmid,NULL,0);
        s = shm;
        printf("Enter message to be sent to child : ");
        scanf("%s",s);
        while(*s!='*')
            sleep(1);
```

```
    }
    else{
        sleep(5);
        printf("In child process :-\n");
        shmid = shmget(key,SHMSZ,0666);
        shm = (char*)shmat(shmid,NULL,0);
        printf("Parent said ");
        for(s = shm;*s!='\0';s++)
            putchar(*s);
        *shm = '*';
    }
}
```

Implementing RR scheduling using pipe

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>

int main() {
    int p1[2], p2[2];
    int ar[20], bt[20], arc[20], btc[20];
    int n;

    pid_t pid;

    if (pipe(p1) == -1) {
        printf("error in creating p1");
        exit(1);
    }

    if (pipe(p2) == -1) {
        printf("error in creating p2");
```

```
    exit(1);
}

pid = fork();

if (pid) {
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    printf("Enter the arrival times and burst times:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &ar[i]);
        scanf("%d", &bt[i]);
    }

    // Send the value of n to the child process
    write(p1[1], &n, sizeof(n));

    // Send arrival times and burst times to the child process
    write(p1[1], ar, sizeof(ar));
    write(p2[1], bt, sizeof(bt));

    sleep(5);
} else {
    sleep(5);

    // Read the value of n from the pipe
    read(p1[0], &n, sizeof(n));

    // Read arrival times and burst times
    read(p1[0], arc, sizeof(arc));
    read(p2[0], btc, sizeof(btc));

    int k = n; // Use the value of n as the number of processes
```

```
int wt[k], tat[k], rt[k];

// Initialize waiting time, turnaround time, and response time arrays
for (int i = 0; i < k; i++) {
    wt[i] = 0;
    tat[i] = 0;
    rt[i] = -1;
}

int rem_t[k];
int cur_ind = 0;
int T_Time = 0;
int quantum = 3;

// Initialize burst time to remaining time
for (int i = 0; i < k; i++) {
    rem_t[i] = btc[i];
}

int com_process=0;


// Implement Round Robin scheduling algorithm
while (com_process < k) {
    if (arc[cur_ind] <= T_Time && rem_t[cur_ind] > 0) {
        if (rt[cur_ind] == -1) {
            rt[cur_ind] = T_Time - arc[cur_ind];
        }

        if (rem_t[cur_ind] > quantum) {
            T_Time += quantum;
            rem_t[cur_ind] -= quantum;
        } else {
            T_Time += rem_t[cur_ind];
            wt[cur_ind] = T_Time - btc[cur_ind];
```

```
                tat[cur_ind] = T_Time - arc[cur_ind];
                rem_t[cur_ind] = 0;
                com_process++;
            }
        }
            cur_ind = (cur_ind + 1) % k;

    }

    printf("Waiting-Time\t\tTurn-
Around-Time\t\tResponse-Time\n");

    for (int i = 0; i < k; i++) {
        printf("%d\t\t\t%d\t\t\t%d\n", wt[i], tat[i], rt[i]);
    }
  }

  return 0;
}
```

Priority Scheduling

```
#include <stdio.h>
struct process
{
        int at;
        int st;
        int pr;
        int status;
        int ft;
}ready_list[10];
int n;
int main()
{
        int i,cur_time,pid;
        printf("Enter number of processes:");
        scanf("%d",&n);
        // Collect process details -
```

```
	for(i=0;i<n;i++)
	{
		printf("Process %d\n",i+1);
		printf("***********\n");
		printf("Enter Arrival Time:");
		scanf("%d",&ready_list[i].at);
		printf("Enter Service Time:");
		scanf("%d",&ready_list[i].st);
		printf("Priority (1-10):");
		scanf("%d",&ready_list[i].pr);
		ready_list[i].status=0;
	}
	i=0; cur_time=0;
	while(i < n)
	{
		pid=dispatcher(cur_time);
		ready_list[pid].ft=cur_time + ready_list[pid].bt;
		ready_list[pid].status=1;
		cur_time+= ready_list[pid].st;
		i++;
	}
	printf("Process\t Arrival Time\t Burst Time\t	Finish Time \t TT \t WT\n");
	printf("*******\t ************\t ************\t ***********\n");
	for(i=0;i<n;i++)
	{

printf("%d\t\t%d\t\t%d\t\t\t%d\n",i,ready_list[i].at,ready_list[i].bt,ready_list[i].ft,
(ready_list[i].ft-ready_list[i].at),(ready_list[i].ft-ready_list[i].at) -ready_list[i].bt);
	}
}
int dispatcher(int time)
{
	int i,high_pr=0,index=-1;
	for(i=0;i<n;i++)
	{
		if(ready_list[i].status != 1)
			if(ready_list[i].at <= time)
				if(ready_list[i].pr > high_pr)
				{
					high_pr = ready_list[i].pr;
```

```
                                        index=i;
                                    }
        }
        return index;
}
```

FIFO Page Replacement:-

```
#include<stdio.h>
void init();
void display();
int N;
struct RAM
{
int pno;
}frame[10];
int q_ptr=0;
int main()
{
int p_no, pf=0;
printf("Enter the number of Frames in memory: ");
scanf("%d",&N);
init();
display();
while(1)
{
printf("\nRequest Page.No: ");
scanf("%d",&p_no);
if(p_no==-1) break;
if(isitinRAM(p_no) != -1)
printf("\nPage Exist");
else
{

pf++;
```

```
printf("\nPage Fault %d", pf);
frame[q_ptr].pno=p_no;
q_ptr=(q_ptr+1)%N;
}
display();
}
return 0;
}
int isitinRAM(int p_no)
{ int i,flag=0;
for(i=0;i<N;i++)
if(frame[i].pno==p_no)
return i;

return -1;
}
void init()
{ int i;
for(i=0;i<N;i++)
frame[i].pno=-1;
}
void display()
{ int i;
printf("\nFrame(RAM)");
for(i=0;i<N;i++)
printf("\n%d",frame[i].pno);

}
```

LFU Page Replacement :-

```
#include <stdio.h>
#include <limits.h>
int n;
```

```
struct RAM{
	int pno;
}frame[10];

int q_ptr=0;

void init(){
	int i;
	for(i=0;i<n;i++)
	frame[i].pno=-1;
}
void display(){
	int i;
	printf("\nFrame(RAM)");
	for(i=0;i<n;i++)
	printf("\n%d",frame[i].pno);
}
int isinRAM(int p_no){
	int i,flag=0;
	for(i=0;i<n;i++){
	if(frame[i].pno==p_no){
	return i;
	}
	}
	return -1;
}


int main(){
	int pf=0;
	int np;
	printf("Enter the number of frames in memory: ");
	scanf("%d",&n);
	printf("Enter the number of page requests:	");
```

```
scanf("%d",&np);
init();
int page_req[np];
printf("\nEnter the Page number requested one by one:     ");
for(int i=0;i<np;i++){
scanf("%d",&page_req[i]);
}
display();
int freq[n];

for(int i=0;i<np;i++){
if(page_req[i]==-1)
break;
if(isinRAM(page_req[i])!=-1){
printf("\nPage Exists...");
}
else{
pf++;
printf("\nPage Fault %d",pf);
if(q_ptr<n){
        frame[q_ptr].pno=page_req[i];
        q_ptr++;
}
else{
        for(int a=0;a<n;a++){
        freq[a]=0;
        }
        for(int j=0;j<n;j++){
        for(int k=i-1;k>=0;k--){
                if(page_req[k]==frame[j].pno){
                freq[j]++;
                }
        }
        }
        int max_occ=freq[0];
```

```
                int process_replace=0;
                for(int j=0;j<n;j++){
                if(freq[j]<max_occ){
                        max_occ=freq[j];
                        process_replace=j;
                }
                }
                frame[process_replace].pno=page_req[i];
        }
        }
        display();
        }
        return 0;
}
```