1) Create two threads thread1 and thread2 and call functions fun1 anf fun2 respectively.

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
void *print_message(void *ptr){
 char *message;
 message=(char*)ptr;
 printf("%s\n",message);
}
int main(){
 pthread_t thread1,thread2;
 char *message1="thread1";
 char *message2="thread2";
 int ir1,ir2;
 ir1=pthread_create(&thread1,NULL,print_message,(void *)message1);
 ir2=pthread_create(&thread2,NULL,print_message,(void *)message2);
 pthread_join(thread1,NULL);
 pthread_join(thread2,NULL);
 printf("Thread 1 returns: %d\n",ir1);
 printf("Thread 2 returns: %d\n",ir2);
 exit(0);
}
```

2) Create two threads thread1 and thread2 and call functions fun1 anf fun2 respectively.
 Compute and print Finbonacci in fun1 and square of a number in fun2.

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
void *fun1(void *ptr){
 int value = *((int*)ptr),n1=0,n2=1,n3;
 printf("Fibonacci series upto %d numbers is :", value);
 printf("%d %d",0,1);
 for(int i=2;i<value; ++i) {
 n3=n1+n2;
 printf("%d ",n3);
 n1=n2;
 n2=n3;
 }
}
void *fun2(void *ptr) {
 int value = *((int *)ptr),sq;
 sq=value*value;
 printf("\nSquare of a %d is = %d\n", value, sq);
}
int main(){
 pthread_t thread1,thread2;
 int n1=12, n2=11;
 int ir1,ir2;
 ir1=pthread_create(&thread1,NULL,fun1,(void*)&n1);
```

```
 ir2=pthread_create(&thread2,NULL,fun2,(void*)&n2);
 pthread_join(thread1,NULL);
 pthread_join(thread2,NULL);
 printf("Thread 1 returns: %d\n",ir1);
 printf("Thread 2 returns: %d\n",ir2);
 exit(0);
}
```

3) Create two threads thread1 and thread2 and call functions fun1 anf fun2 respectively.
 Compute and print Factorial in fun1 and Prime number in fun2.

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
void *fun1(void *ptr){
 int value = *((int*)ptr),factorial=1;
 for(int i=1;i<=value; i++) {
 factorial=factorial*i;
 }
 printf("Factorial of the %d is: %d\n",value,factorial);
}
void *fun2(void *ptr) {
 int n = *((int *)ptr),m=0,flag=0,i;
 m=n/2;
 for(i=2;i<=m;i++)
 {
 if(n%i==0)
 {
 printf("%d is not prime\n",n);
 flag=1;
 break;
 }
 }
 if(flag==0)
 printf("%d is prime\n",n);
}
int main(){
 pthread_t thread1,thread2;
 int n1=5, n2=12;
 int ir1,ir2;
 ir1=pthread_create(&thread1,NULL,fun1,(void*)&n1);
 ir2=pthread_create(&thread2,NULL,fun2,(void*)&n2);
 pthread_join(thread1,NULL);
 pthread_join(thread2,NULL);
 printf("Thread 1 returns: %d\n",ir1);
 printf("Thread 2 returns: %d\n",ir2);
 exit(0);


}
```

4) Create two threads thread1 and thread2 and call functions fun1 anf fun2 respectively.

Compute and print Armstrong number or not in fun1 and Reverse number in fun2.

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
void *fun1(void *ptr){
 int n = *((int*)ptr);
 int k=n;
 int r,sum=0,temp;
 temp=n;
 while(n>0)
 {
 r=n%10;
 sum=sum+(r*r*r);
 n=n/10;
 }
 if(temp==sum)
 printf("%d is an armstrong number\n",k);
 else
 printf("%d is not an armstrong number\n",k);
}
void *fun2(void *ptr) {
 int n = *((int *)ptr);
 int reverse=0, rem;
 while(n!=0)
 {
 rem=n%10;
 reverse=reverse*10+rem;
 n/=10;
 }
 printf("Reversed of %d is: %d\n",n,reverse);
}
int main(){
 pthread_t thread1,thread2;
 int n1=153, n2=1234;
 int ir1,ir2;
 ir1=pthread_create(&thread1,NULL,fun1,(void*)&n1);
 ir2=pthread_create(&thread2,NULL,fun2,(void*)&n2);
 pthread_join(thread1,NULL);
 pthread_join(thread2,NULL);
 printf("Thread 1 returns: %d\n",ir1);
 printf("Thread 2 returns: %d\n",ir2);
 exit(0);

}
```

1) Create a process and Parent ID and Child ID.

CODE:

```
#include <stdio.h>
#include <unistd.h>
int main() {
pid_t child_pid;
```

```
child_pid = fork(); // Create a child process
if (child_pid < 0) {
fprintf(stderr, "Fork failed.\n");
return 1;
}
else if (child_pid == 0) {
// Child process
printf("Child process: PID = %d\n", getpid());
printf("Parent process ID = %d\n", getppid());
}
else {
// Parent process
printf("Parent process: PID = %d\n", getpid());
printf("Child process ID = %d\n", child_pid);
}
return 0;
}
```

2) Create Orphan Process program

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
pid_t pid = fork();
if (pid == 0) {
// Child process
printf("Child process: PID = %d\n", getpid());
printf("Parent process ID: %d\n", getppid());
sleep(5);
printf("New Parent process ID: %d\n", getppid());
} else if (pid > 0) {
// Parent process
printf("Parent process: PID = %d\n", getpid());
exit(0); // Terminate the parent process immediately
} else {
// Error occurred during fork
printf("Fork failed\n");
return 1;
}
return 0;
}
```

3) Write C program using wait system call.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
pid_t pid;
```

```
int status;
pid = fork(); // Create a child process
if (pid < 0) {
// Fork failed
perror("fork");
exit(1);
} else if (pid == 0) {
// Child process
printf("Child process executing\n");
sleep(2); // Simulate some work being done by the child process
exit(0);
} else {
// Parent process
printf("Parent process waiting for child to complete\n");
wait(&status); // Wait for the child process to finish
printf("Child process completed\n");
}
return 0;
}
```

4) Create a process and compute factorial in child and Fibonacci in parent as
executable.

CODE:

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
void ParentProcess(int n){
int t1 = 0, t2 = 1, next = 0, i;
if(n == 0 || n == 1){
printf("The %dth Fibonacci Number is %d\n", n, n);
}
else{
next = t1 + t2;
}
for (i = 3; i <= n; ++i){
t1 = t2;
t2 = next;
next = t1 + t2;
}
printf("The %dth Fibonacci Number is %d\n", n, t2);
}
void ChildProcess(int n){
int ans = 1;
for (int i=1; i<=n; i++){
ans = ans*i;
}
printf("The factorial of %d is %d\n", n, ans);
}
int main(){
pid_t pid;
pid = fork();
int num = 6;
if (pid==0){
```

```
ChildProcess(num);
}
else if (pid>0){
ParentProcess(num);
}
return 1;
}
```

4) Create a process and compute factorial in child and Fibonacci in parent as
executable.
CODE:

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
void ParentProcess(int n){
int t1 = 0, t2 = 1, next = 0, i;
if(n == 0 || n == 1){
printf("The %dth Fibonacci Number is %d\n", n, n);
}
else{
next = t1 + t2;
}
for (i = 3; i <= n; ++i){
t1 = t2;
t2 = next;
next = t1 + t2;
}
printf("The %dth Fibonacci Number is %d\n", n, t2);
}
void ChildProcess(int n){
int ans = 1;
for (int i=1; i<=n; i++){
ans = ans*i;
}
printf("The factorial of %d is %d\n", n, ans);
}
int main(){
pid_t pid;
pid = fork();
int num = 6;
if (pid==0){
ChildProcess(num);
}
else if (pid>0){
ParentProcess(num);
}
return 1;
}
```

6) Palindrome and ODD or EVEN as parent and child with Fork.
CODE:

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

```
#include <sys/wait.h>
#include <sys/types.h>
int isPalindrome(int num) {
int reversedNum = 0, remainder, originalNum;
originalNum = num;
// Reversing the number
while (num != 0) {
remainder = num % 10;
reversedNum = reversedNum * 10 + remainder;
num /= 10;
}
// Checking if the number is a palindrome
if (originalNum == reversedNum)
return 1;
else
return 0;
}
int main() {
pid_t pid;
int num;
printf("Enter a number: ");
scanf("%d", &num);
pid = fork();
if (pid == 0) {
// Child process
int isPal = isPalindrome(num);
if (isPal)
printf("%d is a palindrome.\n", num);
else
printf("%d is not a palindrome.\n", num);
} else if (pid > 0) {
// Parent process
printf("Parent process is waiting for the child to complete...\n");
wait(NULL);
if (num % 2 == 0)
printf("%d is even.\n", num);
else
printf("%d is odd.\n", num);
} else {
// Fork failed
printf("Fork failed.\n");
return 1;
}
return 0;
}
```

1. FCFS Scheduling

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
typedef struct
{
 int pid;
```

```
 int burst_time;
 int waiting_time;
 int turnaround_time;
} Process;
void print_table(Process p[], int n);
void print_gantt_chart(Process p[], int n);
int main()
{
 Process p[MAX];
 int i, j, n;
 int sum_waiting_time = 0, sum_turnaround_time;
 printf("Enter total number of process: ");
 scanf("%d", &n);
 printf("Enter burst time for each process:\n");
 for(i=0; i<n; i++) {
 p[i].pid = i+1;
 printf("P[%d] : ", i+1);
 scanf("%d", &p[i].burst_time);
 p[i].waiting_time = p[i].turnaround_time = 0;
 }
 // calculate waiting time and turnaround time
 p[0].turnaround_time = p[0].burst_time;
 for(i=1; i<n; i++) {
 p[i].waiting_time = p[i-1].waiting_time + p[i-1].burst_time;
 p[i].turnaround_time = p[i].waiting_time + p[i].burst_time;
 }
 // calculate sum of waiting time and sum of turnaround time
 for(i=0; i<n; i++) {
 sum_waiting_time += p[i].waiting_time;
 sum_turnaround_time += p[i].turnaround_time;
 }
 // print table
 puts(""); // Empty line
 print_table(p, n);
 puts(""); // Empty Line
 printf("Total Waiting Time : %-2d\n", sum_waiting_time);
 printf("Average Waiting Time : %-2.2lf\n",
(double)sum_waiting_time / (double) n);
 printf("Total Turnaround Time : %-2d\n", sum_turnaround_time);
 printf("Average Turnaround Time : %-2.2lf\n",
(double)sum_turnaround_time / (double)
n);
 // print Gantt chart
 puts(""); // Empty line
 puts(" GANTT CHART ");
 puts(" ********** ");
 print_gantt_chart(p, n);
 return 0;
}
void print_table(Process p[], int n)
{
 int i;
 puts("+-----+-----------+-------------+-----------------+");
 puts("| PID | Burst Time | Waiting Time | Turnaround Time |");
```

```
 puts("+-----+-----------+-------------+-----------------+");
 for(i=0; i<n; i++) {
 printf("| %2d | %2d | %2d | %2d |\n"
 , p[i].pid, p[i].burst_time, p[i].waiting_time,
p[i].turnaround_time );
 puts("+-----+-----------+-------------+-----------------+");
 }
}
void print_gantt_chart(Process p[], int n)
{
 int i, j;
 // print top bar
 printf(" ");
 for(i=0; i<n; i++) {
 for(j=0; j<p[i].burst_time; j++) printf("--");
 printf(" ");
 }
 printf("\n|");
 // printing process id in the middle
 for(i=0; i<n; i++) {
 for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
 printf("P%d", p[i].pid);
 for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
 printf("|");
 }
 printf("\n ");
 // printing bottom bar
 for(i=0; i<n; i++) {
 for(j=0; j<p[i].burst_time; j++) printf("--");
 printf(" ");
 }
 printf("\n");
 // printing the time line
 printf("0");
 for(i=0; i<n; i++) {
 for(j=0; j<p[i].burst_time; j++) printf(" ");
 if(p[i].turnaround_time > 9) printf("\b"); // backspace : remove 1
space
 printf("%d", p[i].turnaround_time);
 }
 printf("\n");
}
```

2. SJF Scheduling

CODE:

```
#include<stdio.h>
int main()
{
 int n,

process[10],cpu[10],w[10],t[10],At[10],sum_w=0,sum_t=0,i,j,temp=0,te
mp1=0;
 float avg_w, avg_t;
 printf("enter the number of process\n");
```

```
 scanf("%d", &n);
 for(i=0; i<n; i++)
 {
 printf("Enter cpu time of P%d:",i+1);
 scanf("%d", &cpu[i]);
 printf("\n");
 }
 process[0]=1;
 for(i=1; i<n; i++)
 {
 process[i]=i+1;
 }
 for(i=0; i<n; i++)
 {
 for(j=i+1; j<n; j++)
 {
 if(cpu[i]>cpu[j])
 {
 temp=cpu[i];
 cpu[i]=cpu[j];
 cpu[j]=temp;
 temp1=process[i];
 process[i]=process[j];
 process[j]=temp1;
 }
 }
 }
 w[0]=0;
 for(i=1; i<n; i++)
 {
 w[i]=w[i-1]+cpu[i-1];
 }
 for(i=0; i<n; i++)
 {
 sum_w=sum_w+w[i];
 }
for(i=0; i<n; i++)
 {
 t[i]=w[i]+cpu[i];
 sum_t=sum_t+t[i];
 }
 printf("Process--CPU_time--Wait--Turnaround\n");
for(i=0; i<n; i++)
 {
 printf(" P%d \t%d \t%d \t%d",process[i],cpu[i],w[i],t[i]);
 printf("\n");
 }
 avg_w=(float)sum_w/n;
 avg_t=(float)sum_t/n;
 printf("average waiting time=%.2f\n",avg_w);
 printf("average turnaround time=%.2f\n",avg_t);
 printf("\n");

printf("==============================GrandChart==================
```

```
=
=================\n");
 printf("|");
 for(i=0; i<n; i++)
 {
 printf(" P%d |",process[i]);
 }
 printf("\n0");
 for(i=0; i<n; i++)
 {
 printf(" %d",t[i]);
 }}
```

3. Priority Scheduling

CODE:

```
#include <stdio.h>
#define MAX_PROCESSES 10
typedef struct {
 int process_id;
 int burst_time;
 int priority;
} Process;
void priorityScheduling(Process processes[], int n) {
 int total_time = 0;
 int waiting_time[MAX_PROCESSES] = {0};
 int turnaround_time[MAX_PROCESSES] = {0};
 // Calculate waiting time and turnaround time for each process
 for (int i = 0; i < n; i++) {
 waiting_time[i] = total_time;
 total_time += processes[i].burst_time;
 turnaround_time[i] = total_time;
 }
 // Calculate average waiting time and turnaround time
 double avg_waiting_time = 0;
 double avg_turnaround_time = 0;
 for (int i = 0; i < n; i++) {
 avg_waiting_time += waiting_time[i];
 avg_turnaround_time += turnaround_time[i];
 }
 avg_waiting_time /= n;
 avg_turnaround_time /= n;
 // Display the table
 printf("Process\tBurst Time\tPriority\tWaiting Time\tTurnaround
Time\n");
 for (int i = 0; i < n; i++) {
 printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process_id,
processes[i].burst_time,
 processes[i].priority, waiting_time[i], turnaround_time[i]);
 }
 printf("Average Waiting Time: %.2f\n", avg_waiting_time);
 printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
 // Display the Gantt chart
 printf("\nGantt Chart:\n");
```

```
 for (int i = 0; i < n; i++) {
 printf("| P%d ", processes[i].process_id);
 }
 printf("|\n");
 printf("0");
 for (int i = 0; i < n; i++) {
 for (int j = 0; j < processes[i].burst_time; j++) {
 printf(" ");
 }
 printf("%2d", turnaround_time[i]);
 }
 printf("\n");
}
int main() {
 int n;
 Process processes[MAX_PROCESSES];
 printf("Enter the number of processes: ");
 scanf("%d", &n);
 printf("Enter burst time and priority for each process:\n");
 for (int i = 0; i < n; i++) {
 processes[i].process_id = i + 1;
 printf("Process %d:\n", processes[i].process_id);
 printf("Burst Time: ");
 scanf("%d", &processes[i].burst_time);
 printf("Priority: ");
 scanf("%d", &processes[i].priority);
 }
 priorityScheduling(processes, n);
 return 0;
}e <stdio.h>
#define MAX_PROCESSES 10
typedef struct {
 int process_id;
 int burst_time;
 int priority;
} Process;
void priorityScheduling(Process processes[], int n) {
 int total_time = 0;
 int waiting_time[MAX_PROCESSES] = {0};
 int turnaround_time[MAX_PROCESSES] = {0};
 // Calculate waiting time and turnaround time for each process
 for (int i = 0; i < n; i++) {
 waiting_time[i] = total_time;
 total_time += processes[i].burst_time;
 turnaround_time[i] = total_time;
 }
 // Calculate average waiting time and turnaround time
 double avg_waiting_time = 0;
 double avg_turnaround_time = 0;
 for (int i = 0; i < n; i++) {
 avg_waiting_time += waiting_time[i];
 avg_turnaround_time += turnaround_time[i];
 }
 avg_waiting_time /= n;
```

```
 avg_turnaround_time /= n;
 // Display the table
 printf("Process\tBurst Time\tPriority\tWaiting Time\tTurnaround
Time\n");
 for (int i = 0; i < n; i++) {
 printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process_id,
processes[i].burst_time,
 processes[i].priority, waiting_time[i], turnaround_time[i]);
 }
 printf("Average Waiting Time: %.2f\n", avg_waiting_time);
 printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
 // Display the Gantt chart
 printf("\nGantt Chart:\n");
 for (int i = 0; i < n; i++) {
 printf("| P%d ", processes[i].process_id);
 }
 printf("|\n");
 printf("0");
 for (int i = 0; i < n; i++) {
 for (int j = 0; j < processes[i].burst_time; j++) {
 printf(" ");
 }
 printf("%2d", turnaround_time[i]);
 }
 printf("\n");
}
int main() {
 int n;
 Process processes[MAX_PROCESSES];
 printf("Enter the number of processes: ");
 scanf("%d", &n);
 printf("Enter burst time and priority for each process:\n");
 for (int i = 0; i < n; i++) {
 processes[i].process_id = i + 1;
 printf("Process %d:\n", processes[i].process_id);
 printf("Burst Time: ");
 scanf("%d", &processes[i].burst_time);
 printf("Priority: ");
 scanf("%d", &processes[i].priority);
 }
 priorityScheduling(processes, n);
 return 0;
}
```

4. Round Robin Scheduling

CODE:

```
#include<stdio.h>
struct times
{
 int p,art,but,wtt,tat,rnt;
};
void sortart(struct times a[],int pro)
{
 int i,j;
```

```
 struct times temp;
 for(i=0;i<pro;i++)
 {
 for(j=i+1;j<pro;j++)
 {
 if(a[i].art > a[j].art)
 {
 temp = a[i];
 a[i] = a[j];
 a[j] = temp;
 }}}
 return;
}
int main()
{
 int i,j,pro,time,remain,flag=0,ts;
 struct times a[100];
 float avgwt=0,avgtt=0;
 printf("Round Robin Scheduling Algorithm\n");
 printf("Note -\n1. Arrival Time of at least on process should be
0\n2. CPU should never be idle\n");
 printf("Enter Number Of Processes : ");
 scanf("%d",&pro);
 remain=pro;
 for(i=0;i<pro;i++)
 {
 printf("Enter arrival time and Burst time for Process P%d : ",i);
 scanf("%d%d",&a[i].art,&a[i].but);
 a[i].p = i;
 a[i].rnt = a[i].but;
 }
 sortart(a,pro);
 printf("Enter Time Slice OR Quantum Number : ");
 scanf("%d",&ts);
 printf("\n**************************************\n");
 printf("Gantt Chart\n");
 printf("0");
 for(time=0,i=0;remain!=0;)
 {
 if(a[i].rnt<=ts && a[i].rnt>0)
 {
 time = time + a[i].rnt;
 printf(" -> [P%d] <- %d",a[i].p,time);
 a[i].rnt=0;
 flag=1;
 }
 else if(a[i].rnt > 0)
 {
 a[i].rnt = a[i].rnt - ts;
 time = time + ts;
 printf(" -> [P%d] <- %d",a[i].p,time);
 }
 if(a[i].rnt==0 && flag==1)
 {
```

```
 remain--;
 a[i].tat = time-a[i].art;
 a[i].wtt = time-a[i].art-a[i].but;
 avgwt = avgwt + time-a[i].art-a[i].but;
 avgtt = avgtt + time-a[i].art;
 flag=0;
 }
 if(i==pro-1)
 i=0;
 else if(a[i+1].art <= time)
 i++;
 else
 i=0;
 }
 printf("\n\n");
 printf("****************************************\n");
 printf("Pro\tArTi\tBuTi\tTaTi\tWtTi\n");
 printf("****************************************\n");
 for(i=0;i<pro;i++)
 {

printf("P%d\t%d\t%d\t%d\t%d\n",a[i].p,a[i].art,a[i].but,a[i].tat,a[i
].wtt);
 }
 printf("****************************************\n");
 avgwt = avgwt/pro;
 avgtt = avgtt/pro;
 printf("Average Waiting Time : %.2f\n",avgwt);
 printf("Average Turnaround Time : %.2f\n",avgtt);
 return 0;
}
```

1. Deadlock Detection

CODE:

```
#include <stdio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n, m, i, j, k;
void input() {
 printf("Enter the no of Processes: ");
 scanf("%d", & n);
 printf("Enter the no of resource instances: ");
 scanf("%d", & m);
 printf("Enter the Max Matrix\n");
 for (i = 0; i < n; i++) {
 for (j = 0; j < m; j++) {
 scanf("%d", & max[i][j]);
 }
 }
 printf("Enter the Allocation Matrix\n");
 for (i = 0; i < n; i++) {
 for (j = 0; j < m; j++) {
```

```
 scanf("%d", & alloc[i][j]);
 }
 }
 printf("Enter the available Resources\n");
 for (j = 0; j < m; j++) {
 scanf("%d", & avail[j]);
 }
}
void show() {
 int i, j;
 printf("Process\t Allocation\t Max\t Available\t");
 for (i = 0; i < n; i++) {
 printf("\nP%d\t ", i + 1);
 for (j = 0; j < m; j++) {
 printf("%d ", alloc[i][j]);
 }
 printf("\t\t");
 for (j = 0; j < m; j++) {
 printf("%d ", max[i][j]);
 }
 printf("\t ");
 if (i == 0) {
 for (j = 0; j < m; j++) {
 printf("%d ", avail[j]);
 }
 }
 }
}
void printTotalAvailableResources() {
 int totalAvailable[100] = {
 0
 };
 for (i = 0; i < m; i++) {
 for (j = 0; j < n; j++) {
 totalAvailable[i] += alloc[j][i];
 }
 totalAvailable[i] = avail[i] + totalAvailable[i];
 }
 printf("\n\nTotal Available Resources: ");
 for (i = 0; i < m; i++) {
 printf("%d ", totalAvailable[i]);
 }
 printf("\n");
}
int main() {
 printf("********** Deadlock Detection Algorithm ***********\n");
 input();
 show();
 int f[n], ans[n], ind = 0;
 for (k = 0; k < n; k++) {
 f[k] = 0;
 }
 int need[n][m];
 for (i = 0; i < n; i++) {
```

```
 for (j = 0; j < m; j++) {
 need[i][j] = max[i][j] - alloc[i][j];
 }
 }
 int y = 0;
 for (k = 0; k < 5; k++) {
 for (i = 0; i < n; i++) {
 if (f[i] == 0) {
 int flag = 0;
 for (j = 0; j < m; j++) {
 if (need[i][j] > avail[j]) {
 flag = 1;
 break;
 }
 }
 if (flag == 0) {
 ans[ind++] = i;
 for (y = 0; y < m; y++) {
 avail[y] += alloc[i][y];
 }
 f[i] = 1;
 }
 }
 }
 }
 int flag = 1;
 for (int i = 0; i < n; i++) {
 if (f[i] == 0) {
 flag = 0;
 printf("\nDeadlock detected!\n");
 printf("The following system is not safe\n");
 break;
 }
 }
 if (flag == 1) {
 printf("\nFollowing is the SAFE Sequence\n");
 for (i = 0; i < n - 1; i++) {
 printf("P%d -> ", ans[i]);
 }
 printf("P%d\n", ans[n - 1]);
 }
 printTotalAvailableResources();
 return 0;
}
```

1. Producer Consumer Problem Using Semaphores

CODE:

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty,x=0;
char a[100];
int main()
{
int n;
```

```
printf("Enter Buffer Size: ");
scanf("%d",&empty);
int j=empty;
void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n1.PRODUCER\n2.CONSUMER\n3.DISPLAY\n4.EXIT\n");
while(1) {
printf("\nENTER YOUR CHOICE\n");
scanf("%d",&n);
switch(n)
{ case 1:
if((mutex==1)&&(empty!=0)){
printf("\nWhat to Produce\n");
scanf("%s",a);
producer(a);
}
else
printf("BUFFER IS FULL\n");
break;
case 2:
if((mutex==1)&&(full!=0))
consumer(a);
else
printf("BUFFER IS EMPTY\n");
break;
case 3:
printf("Buffer Size: %d\n",j-empty);
break;
case 4:
exit(0);
break;
}}}
int wait(int s) {
return(--s); }
int signal(int s) {
return(++s); }
void producer(char a[]) {
mutex=wait(mutex);
full=signal(full);
empty=wait(empty);
x++;
printf("producer produces item%d: %s\n",x,a);
mutex=signal(mutex); }
void consumer(char a[]) {
mutex=wait(mutex);
full=wait(full);
empty=signal(empty);
printf("consumer consumes item%d: %s\n",x,a);
x--;
mutex=signal(mutex);
}
```

2. Readers Writers Problem Using Semaphores

CODE:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
sem_t mutex; // Semaphore for mutual exclusion
sem_t db; // Semaphore for controlling database access
int readercount = 0; // Counter for the number of active readers
void *reader(void *arg);
void *writer(void *arg);
int main()
{
 pthread_t reader1, reader2, writer1, writer2;
 sem_init(&mutex, 0, 1);
 sem_init(&db, 0, 1);
 // Create reader and writer threads
 pthread_create(&reader1, NULL, reader, (void *)1);
 pthread_create(&reader2, NULL, reader, (void *)2);
 pthread_create(&writer1, NULL, writer, (void *)1);
 pthread_create(&writer2, NULL, writer, (void *)2);
 // Wait for threads to finish
 pthread_join(reader1, NULL);
 pthread_join(reader2, NULL);
 pthread_join(writer1, NULL);
 pthread_join(writer2, NULL);
 sem_destroy(&mutex);
 sem_destroy(&db);
 return 0;
}
void *reader(void *arg)
{
 int readerID = (int)arg;
 while (1)
 {
 // Acquire mutex to update reader count
 sem_wait(&mutex);
 readercount++;
 if (readercount == 1)
 {
 // First reader, acquire database
 sem_wait(&db);
 }
 sem_post(&mutex);
 // Reader reading
 printf("Reader %d is reading\n", readerID);
 // Reading is performed here
 // Acquire mutex to update reader count
 sem_wait(&mutex);
 readercount--;
 if (readercount == 0)
 {
 // Last reader, release database
 sem_post(&db);
 }
```

```
 sem_post(&mutex);
 // Reader completes reading
 printf("Reader %d completed reading\n", readerID);
 // Sleep for a while before next read
 sleep(1);
 }
 pthread_exit(NULL);
}
void *writer(void *arg)
{
 int writerID = (int)arg;
 while (1)
 {
 // Writer waiting
 printf("Writer %d is waiting\n", writerID);
 // Acquire database
 sem_wait(&db);
 // Writer writing
 printf("Writer %d is writing\n", writerID);
 // Writing is performed here
 // Release database
 sem_post(&db);
 // Writer completes writing
 printf("Writer %d completed writing\n", writerID);
 // Sleep for a while before next write
 sleep(1);
 }
 pthread_exit(NULL);
}
```

1. FIRST FIT:

CODE:

```
#include<stdio.h>
#define max 25
int main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
static int bf[max],ff[max];
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
```

```
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1) //if bf[j] is not allocated
{
temp=b[j]-f[i];
if(temp>=0)
if(highest<temp)
{
ff[i]=j;
highest=temp;
}
}
}
frag[i]=highest;
bf[ff[i]]=1;
highest=0;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:
\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i])
;
}
```

2. BEST FIT:

CODE:

```
#include<stdio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
static int bf[max],ff[max];
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
printf("Block %d:",i);
scanf("%d",&b[i]);
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
```

```
temp=b[j]-f[i];
if(temp>=0)
if(lowest>temp)
{
ff[i]=j;
lowest=temp;
}
}}
frag[i]=lowest; bf[ff[i]]=1; lowest=10000;
}
printf("\nFile No\tFile Size \tBlock No\tBlock
Size\tFragment"); for(i=1;i<=nf && ff[i]!=0;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i])
;
}
```

3. WORST FIT:

CODE:

```
#include<stdio.h>
#define max 25
int main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp;
static int bf[max],ff[max];
printf("\n\tMemory Management Scheme - First Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
{
ff[i]=j;
break;
}
}
```

```
}
frag[i]=temp;
bf[ff[i]]=1;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:
\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i])
;
}
```

1. FIFO:

CODE:

```
// C program for FIFO page replacement algorithm
#include<stdio.h>
int main()
{
int incomingStream[] = {1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1,
3};
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;
}
```

2. LRU:

CODE:

```
// C program for LRU page replacement algorithm
#include<stdio.h>
#include<limits.h>
int checkHit(int incomingPage, int queue[], int occupied){
for(int i = 0; i < occupied; i++){
if(incomingPage == queue[i])
return 1;
}
return 0;
}
void printFrame(int queue[], int occupied)
{
for(int i = 0; i < occupied; i++)
printf("%d\t\t\t",queue[i]);
}
int main()
{
// int incomingStream[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,
1};
// int incomingStream[] = {1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1,
3, 6, 1, 2, 4, 3};
int incomingStream[] = {1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1,
3};
int n = sizeof(incomingStream)/sizeof(incomingStream[0]);
int frames = 3;
int queue[n];
int distance[n];
int occupied = 0;
int pagefault = 0;
printf("Page\t Frame1 \t Frame2 \t Frame3\n");
for(int i = 0;i < n; i++)
{
printf("%d: \t\t",incomingStream[i]);
// what if currently in frame 7
// next item that appears also 7
// didnt write condition for HIT
if(checkHit(incomingStream[i], queue, occupied)){
printFrame(queue, occupied);
}
// filling when frame(s) is/are empty
else if(occupied < frames){
```

```
queue[occupied] = incomingStream[i];
pagefault++;
occupied++;
printFrame(queue, occupied);
}
else{
int max = INT_MIN;
int index;
// get LRU distance for each item in frame
for (int j = 0; j < frames; j++)
{
distance[j] = 0;
// traverse in reverse direction to find
// at what distance frame item occurred last
for(int k = i - 1; k >= 0; k--)
{
++distance[j];
if(queue[j] == incomingStream[k])
break;
}
// find frame item with max distance for LRU
// also notes the index of frame item in queue
// which appears furthest(max distance)
if(distance[j] > max){
max = distance[j];
index = j;
}
}
queue[index] = incomingStream[i];
printFrame(queue, occupied);
pagefault++;
}
printf("\n");
}
printf("Page Fault: %d",pagefault);
return 0;
}
```

3. OPTIMAL:

CODE:

```
#include<stdio.h>
int main()
{
 int no_of_frames=3, no_of_pages=15, frames[10], temp[10], flag1,
flag2, flag3, i, j, k, pos,
max, faults = 0;


 int pages[30]={1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1, 3};

 for(i = 0; i < no_of_frames; ++i){
 frames[i] = -1;
 }
```

```
for(i = 0; i < no_of_pages; ++i){
flag1 = flag2 = 0;

for(j = 0; j < no_of_frames; ++j){
if(frames[j] == pages[i]){
flag1 = flag2 = 1;
break;
}
}

if(flag1 == 0){
for(j = 0; j < no_of_frames; ++j){
if(frames[j] == -1){
faults++;
frames[j] = pages[i];
flag2 = 1;
break;
}
}
}

if(flag2 == 0){
flag3 =0;

for(j = 0; j < no_of_frames; ++j){
temp[j] = -1;

for(k = i + 1; k < no_of_pages; ++k){
if(frames[j] == pages[k]){
temp[j] = k;
break;
}
}
}

for(j = 0; j < no_of_frames; ++j){
if(temp[j] == -1){
pos = j;
flag3 = 1;
break;
}
}

if(flag3 ==0){
max = temp[0];
pos = 0;

for(j = 1; j < no_of_frames; ++j){
if(temp[j] > max){
max = temp[j];
pos = j;
}
}
}
```

```
frames[pos] = pages[i];
faults++;
 }

 printf("\n");

 for(j = 0; j < no_of_frames; ++j){
 printf("%d\t", frames[j]);
 }
 }

 printf("\n\nTotal Page Faults = %d", faults);

 return 0;
}
```