VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT

on

Operating Systems

(22CS4PCOPS)

Submitted by:

Bhoomi Udedh(1BM23CS066)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Feb 2025 - June 2025

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Operating Systems**" carried out by **Bhoomi Udedh(1BM23CS066),** who is a bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024-25. The Lab report has been approved as it satisfies the academic requirements in respect of **Operating Systems - (22CS4PCOPS)** work prescribed for the said degree.

Basavaraj Jakkalli
Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Table Of Contents

Course Outcomes

CO1: Apply the different concepts and functionalities of Operating System.

CO2: Analyse various Operating system strategies and techniques.

CO3: Demonstrate the different functionalities of Operating System.

CO4: Conduct practical experiments to implement the functionalities of Operating system.

GITHUB LINK:

https://github.com/bhoomiudedh/OS

Experiments

1. **Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.**

(a) FCFS

(b) SJF

```
#include<stdio.h>
   int n, i, j, pos, temp, choice, Burst_time[20], Waiting_time[20],
   Turn_around_time[20], process[20], total=0;
   float avg_Turn_around_time=0, avg_Waiting_time=0;

int FCFS()
{
   Waiting_time[0]=0;

   for(i=1;i<n;i++)
   {
      Waiting_time[i]=0;
      for(j=0;j<i;j++)
         Waiting_time[i]+=Burst_time[j];
   }

   printf("\nProcess\t\tBurst Time\t\tWaiting Time\t\tTurnaround Time");

   for(i=0;i<n;i++)
   {
      Turn_around_time[i]=Burst_time[i]+Waiting_time[i];
      avg_Waiting_time+=Waiting_time[i];
      avg_Turn_around_time+=Turn_around_time[i];

printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t\t%d",i+1,Burst_time[i],Waiting_time[i],Turn_around_t
ime[i]);
   }

   avg_Waiting_time =(float)(avg_Waiting_time)/(float)i;
   avg_Turn_around_time=(float)(avg_Turn_around_time)/(float)i;
   printf("\nAverage Waiting Time:%.2f",avg_Waiting_time);
   printf("\nAverage Turnaround Time:%.2f\n",avg_Turn_around_time);

   return 0;
}
```

```
int SJF()
{
   //sorting
   for(i=0;i<n;i++)
   {
      pos=i;
      for(j=i+1;j<n;j++)
      {
         if(Burst_time[j]<Burst_time[pos])
            pos=j;
      }

      temp=Burst_time[i];
      Burst_time[i]=Burst_time[pos];
      Burst_time[pos]=temp;

      temp=process[i];
      process[i]=process[pos];
      process[pos]=temp;
   }
      Waiting_time[0]=0;


   for(i=1;i<n;i++)
   {
      Waiting_time[i]=0;

      for(j=0;j<i;j++)
         Waiting_time[i]+=Burst_time[j];

      total+=Waiting_time[i];
   }

   avg_Waiting_time=(float)total/n;
   total=0;

   printf("\nProcess\t\tBurst Time\t\tWaiting Time\t\tTurnaround Time");

   for(i=0;i<n;i++)
   {
      Turn_around_time[i]=Burst_time[i]+Waiting_time[i];
      total+=Turn_around_time[i];

printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t\t%d",process[i],Burst_time[i],Waiting_time[i],Turn_ar
ound_time[i]);
```

```
    }

    avg_Turn_around_time=(float)total/n;
    printf("\n\nAverage Waiting Time=%f",avg_Waiting_time);
    printf("\nAverage Turnaround Time=%f\n",avg_Turn_around_time);
}

int main()
{
    printf("Enter the total number of processes:");
    scanf("%d",&n);

    printf("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("P[%d]:",i+1);
        scanf("%d",&Burst_time[i]);
        process[i]=i+1;
    }

    while(1)
    {   printf("\n-----MAIN MENU -----\n");
        printf("1. FCFS Scheduling\n2. SJF Scheduling\n");
        printf("\nEnter your choice:");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1: FCFS();
            break;

            case 2: SJF();
            break;

            default: printf("Invalid Input!!!");
        }
    }
    return 0;
}
```

Output:

a.

```
ArrivalTime.c -o FCFS_ArrivalTime } ; if ($?) { .\FCFS_ArrivalTime }
Enter the number of processes: 4
Enter the process ids:
1 2 3 4
Enter arrival time and burst time for process 1: 0 8
Enter arrival time and burst time for process 2: 1 4
Enter arrival time and burst time for process 3: 2 9
Enter arrival time and burst time for process 4: 3 5

Process Arrival Time    Burst Time    Waiting Time    Turnaround Time
1       0               8             0               8
2       1               4             7               11
3       2               9             10              19
4       3               5             18              23

Average Waiting Time: 8.75
Average Turnaround Time: 15.25
PS C:\Users\Nisarga Gondi\OneDrive\Desktop\Nisarga\lV SEM\OS 4th sem\os lab>
```

b.

```
P.c -o SJF_NP } ; if ($?) { .\SJF_NP }
Enter the number of processes:
4
Enter the burst time of process 1:
8
Enter the burst time of process 2:
4
Enter the burst time of process 3:
9
Enter the burst time of process 4:
5
BurstTime      WaitingTime     TurnAroundtime
4.00           0.00            4.00
5.00           4.00            9.00
8.00           9.00            17.00
9.00           17.00           26.00
Average waiting time:7.500000
Average turn around time:14.000000
```

2.Priority and Round Robin

```
#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]; p
      ri[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();
```

```
}
#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;
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();
}
```

Output:

3. Rate Monotonic and Earliest Deadline first

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

#define MAX_PROCESS 10

int num_of_process;
int execution_time[MAX_PROCESS], period[MAX_PROCESS],
remain_time[MAX_PROCESS];

int max(int a, int b, int c) {
   if (a >= b && a >= c) return a;
   if (b >= a && b >= c) return b;
   return c;
}

void get_process_info() {
   printf("Enter total number of processes (maximum %d): ", MAX_PROCESS);
   scanf("%d", &num_of_process);
   if (num_of_process < 1) {
      printf("Invalid number of processes.\n");
      exit(0);
   }
   for (int i = 0; i < num_of_process; i++) {
      printf("\nProcess %d:\n", i + 1);
      printf("==> Execution time: ");
      scanf("%d", &execution_time[i]);
      remain_time[i] = execution_time[i];
      printf("==> Period: ");
      scanf("%d", &period[i]);
   }
}

int get_observation_time() {
   int max_period = 0;
   for (int i = 0; i < num_of_process; i++) {
      if (period[i] > max_period)
```

```
        max_period = period[i];
    }
    return max_period;
}

void print_schedule(int process_list[], int cycles) {
    printf("\nScheduling:\n\nTime: ");
    for (int i = 0; i < cycles; i++) {
        if (i < 10) printf("| 0%d ", i);
        else printf("| %d ", i);
    }
    printf("|\n");

    for (int i = 0; i < num_of_process; i++) {
        printf("P[%d]: ", i + 1);
        for (int j = 0; j < cycles; j++) {
            if (process_list[j] == i + 1) printf("|####");
            else printf("|    ");
        }
        printf("|\n");
    }
}

void rate_monotonic(int time) {
    int process_list[100] = {0}, min = 999, next_process = 0;
    float utilization = 0;
    for (int i = 0; i < num_of_process; i++) {
        utilization += (1.0 * execution_time[i]) / period[i];
    }

    int n = num_of_process;
    int m = (int)(n * (pow(2, 1.0 / n) - 1));
    if (utilization > m) {
            printf("\nGiven problem is not schedulable under the Rate Monotonic
algorithm.\n");
    }

    for (int i = 0; i < time; i++) {
        min = 1000;
        for (int j = 0; j < num_of_process; j++) {
            if (remain_time[j] > 0 && min > period[j]) {
                min = period[j];
                next_process = j;
            }
        }
```

```
        if (remain_time[next_process] > 0) {
            process_list[i] = next_process + 1;
            remain_time[next_process] -= 1;
        }
        for (int k = 0; k < num_of_process; k++) {
            if ((i + 1) % period[k] == 0) {
                remain_time[k] = execution_time[k];
            }
        }
    }
    print_schedule(process_list, time);
}

int main() {
    get_process_info();
    int observation_time = get_observation_time();
    rate_monotonic(observation_time);
    return 0;
}

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

#define MAX_PROCESS 10

int num_of_process;
int execution_time[MAX_PROCESS], deadline[MAX_PROCESS],
remain_time[MAX_PROCESS], remain_deadline[MAX_PROCESS];

int max(int a, int b, int c) {
    if (a >= b && a >= c) return a;
    if (b >= a && b >= c) return b;
    return c;
}

void get_process_info() {
    printf("Enter total number of processes (maximum %d): ", MAX_PROCESS);
    scanf("%d", &num_of_process);
    if (num_of_process < 1) {
        printf("Invalid number of processes.\n");
        exit(0);
    }
    for (int i = 0; i < num_of_process; i++) {
        printf("\nProcess %d:\n", i + 1);
```

```
        printf("==> Execution time: ");
        scanf("%d", &execution_time[i]);
        remain_time[i] = execution_time[i];
        printf("==> Deadline: ");
        scanf("%d", &deadline[i]);
    }
}

int get_observation_time() {
    int max_deadline = 0;
    for (int i = 0; i < num_of_process; i++) {
        if (deadline[i] > max_deadline)
            max_deadline = deadline[i];
    }
    return max_deadline;
}

void print_schedule(int process_list[], int cycles) {
    printf("\nScheduling:\n\nTime: ");
    for (int i = 0; i < cycles; i++) {
        if (i < 10) printf("| 0%d ", i);
        else printf("| %d ", i);
    }
    printf("|\n");

    for (int i = 0; i < num_of_process; i++) {
        printf("P[%d]: ", i + 1);
        for (int j = 0; j < cycles; j++) {
            if (process_list[j] == i + 1) printf("|####");
            else printf("|    ");
        }
        printf("|\n");
    }
}

void earliest_deadline_first(int time) {
    float utilization = 0;
    for (int i = 0; i < num_of_process; i++) {
        utilization += (1.0 * execution_time[i]) / deadline[i];
    }

    int process[num_of_process];
    int max_deadline, current_process = 0, min_deadline, process_list[time];
    bool is_ready[num_of_process];
    for (int i = 0; i < num_of_process; i++) {
```

```
    is_ready[i] = true;
    process[i] = i + 1;
}

max_deadline = deadline[0];
for (int i = 1; i < num_of_process; i++) {
    if (deadline[i] > max_deadline) max_deadline = deadline[i];
}

// Sorting by deadline
for (int i = 0; i < num_of_process; i++) {
    for (int j = i + 1; j < num_of_process; j++) {
        if (deadline[j] < deadline[i]) {
            int temp = execution_time[j];
            execution_time[j] = execution_time[i];
            execution_time[i] = temp;
            temp = deadline[j];
            deadline[j] = deadline[i];
            deadline[i] = temp;
            temp = process[j];
            process[j] = process[i];
            process[i] = temp;
        }
    }
}

for (int i = 0; i < num_of_process; i++) {
    remain_time[i] = execution_time[i];
    remain_deadline[i] = deadline[i];
}

for (int t = 0; t < time; t++) {
    if (current_process != -1) {
        --execution_time[current_process];
        process_list[t] = process[current_process];
    } else process_list[t] = 0;

    for (int i = 0; i < num_of_process; i++) {
        --deadline[i];
        if ((execution_time[i] == 0) && is_ready[i]) {
            deadline[i] += remain_deadline[i];
            is_ready[i] = false;
        }
        if ((deadline[i] <= remain_deadline[i]) && !is_ready[i]) {
            execution_time[i] = remain_time[i];
```

```
                is_ready[i] = true;
            }
        }

        min_deadline = max_deadline;
        current_process = -1;
        for (int i = 0; i < num_of_process; i++) {
            if ((deadline[i] <= min_deadline) && (execution_time[i] > 0)) {
                current_process = i;
                min_deadline = deadline[i];
            }
        }
    }
    print_schedule(process_list, time);
}

int main() {
    get_process_info();
    int observation_time = get_observation_time();
    earliest_deadline_first(observation_time);
    return 0;
}
```

Output:

```
1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice: 1
Enter total number of processes (maximum 10): 3

Process 1:
==> Execution time: 3
==> Period: 20

Process 2:
--> Execution time: 2
--> Period: 5

Process 3:
==> Execution time: 2
==> Period: 10

Scheduling:

Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
P[1]: |    |    |    |    |####|    |    |####|####|    |    |    |    |    |    |    |    |    |    |    |
P[2]: |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |
P[3]: |    |    |####|####|    |    |    |    |    |    |    |    |####|####|    |    |    |    |    |    |
```

```
1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice: 2
Enter total number of processes (maximum 10): 3

Process 1:
==> Execution time: 3
==> Deadline: 7

Process 2:
==> Execution time: 2
==> Deadline: 4

Process 3:
==> Execution time: 2
==> Deadline: 8

Scheduling:

Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
P[1]: |    |    |####|####|####|    |    |    |
P[2]: |####|####|    |    |    |    |    |####|
P[3]: |    |    |    |    |    |####|####|    |
```

4. Producer Consumer and Dining Philosophers

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

int mutex = 1, full = 0, empty = 3, x = 0;

void producer();
void consumer();
int wait(int);
int signal(int);

int main() {
    int n;
    printf("\n1. Producer\n2. Consumer\n3. Exit");
    while(1) {
        printf("\nEnter your choice: ");
        scanf("%d", &n);
        switch(n) {
            case 1:
                if((mutex == 1) && (empty != 0))
                    producer();
```

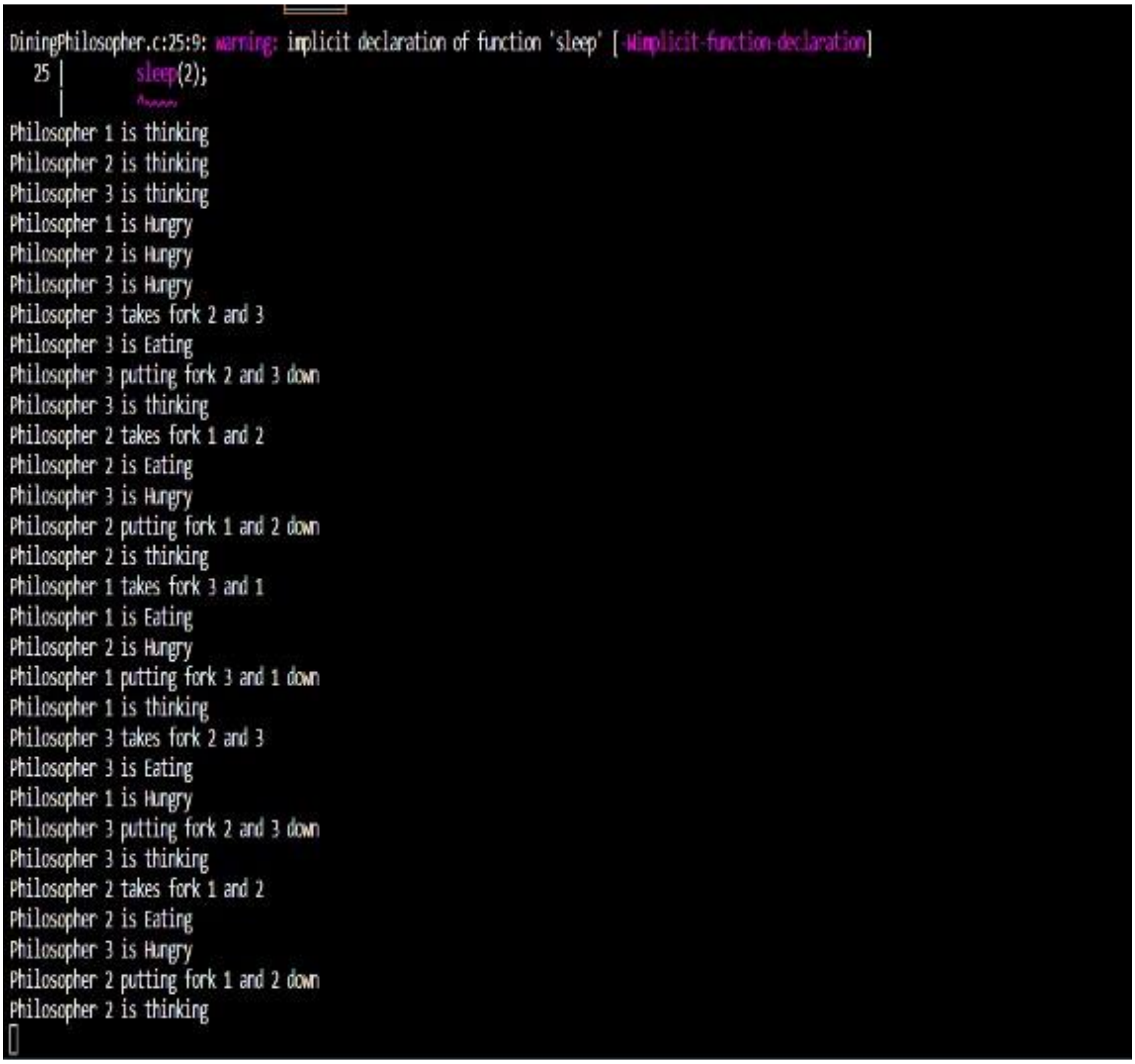
```
            else
                printf("Buffer is full!!\n");
            break;
        case 2:
            if((mutex == 1) && (full != 0))
                consumer();
            else
                printf("Buffer is empty!!\n");
            break;
        case 3:
            exit(0);
            break;
        }
    }
    return 0;
}

int wait(int s) {
    return (--s);
}

int signal(int s) {
    return (++s);
}

void producer() {
    mutex = wait(mutex);
    full = signal(full);
    empty = wait(empty);
    x++;
    printf("\nProducer produces the item %d\n", x);
    mutex = signal(mutex);
}

void consumer() {
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
    printf("\nConsumer consumes item %d\n", x);
    x--;
    mutex = signal(mutex);
}

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

```
#include <semaphore.h>
#include <unistd.h>  // For usleep

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (i + 4) % N
#define RIGHT (i + 1) % N

int state[N];
int phil[N] = {0, 1, 2, 3, 4};
sem_t mutex;
sem_t S[N];

void test(int i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        usleep(2000000);  // Simulate eating time (2 seconds)
        printf("Philosopher %d takes fork %d and %d\n", i + 1, LEFT + 1, i + 1);
        printf("Philosopher %d is Eating\n", i + 1);
        sem_post(&S[i]);
    }
}

void take_fork(int i) {
    sem_wait(&mutex);
    state[i] = HUNGRY;
    printf("Philosopher %d is Hungry\n", i + 1);
    test(i);
    sem_post(&mutex);
    sem_wait(&S[i]);
    usleep(1000000);  // Simulate thinking time (1 second)
}

void put_fork(int i) {
    sem_wait(&mutex);
    state[i] = THINKING;
    printf("Philosopher %d putting fork %d and %d down\n", i + 1, LEFT + 1, i + 1);
    printf("Philosopher %d is thinking\n", i + 1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}
```

```
void* philosopher(void* num) {
    while (1) {
        int* i = num;
        usleep(1000000);  // Simulate thinking before trying to eat
        take_fork(*i);
        usleep(1000000);  // Simulate time spent eating
        put_fork(*i);
    }
}

int main() {
    int i;
    pthread_t thread_id[N];
    sem_init(&mutex, 0, 1);
    for (i = 0; i < N; i++) {
        sem_init(&S[i], 0, 0);
    }

    for (i = 0; i < N; i++) {
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }

    for (i = 0; i < N; i++) {
        pthread_join(thread_id[i], NULL);
    }

    return 0;
}
```

Output:

```
cerConcumer.c -o ProducerConcumer } ; if ($?) { .\ProducerConcumer }
Produced: 0
Produced: 1
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Produced: 6
Produced: 7
Produced: 8
Produced: 9
Consumed: 0
Consumed: 1
Consumed: 2
Consumed: 3
Consumed: 4
Consumed: 5
Consumed: 6
Consumed: 7
Consumed: 8
Consumed: 9
```



```
DiningPhilosopher.c:25:9: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
   25 |         sleep(2);
      |         ^~~~~
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 3 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 3 and 1
Philosopher 1 is Eating
Philosopher 2 is Hungry
Philosopher 1 putting fork 3 and 1 down
Philosopher 1 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 1 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 3 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
```

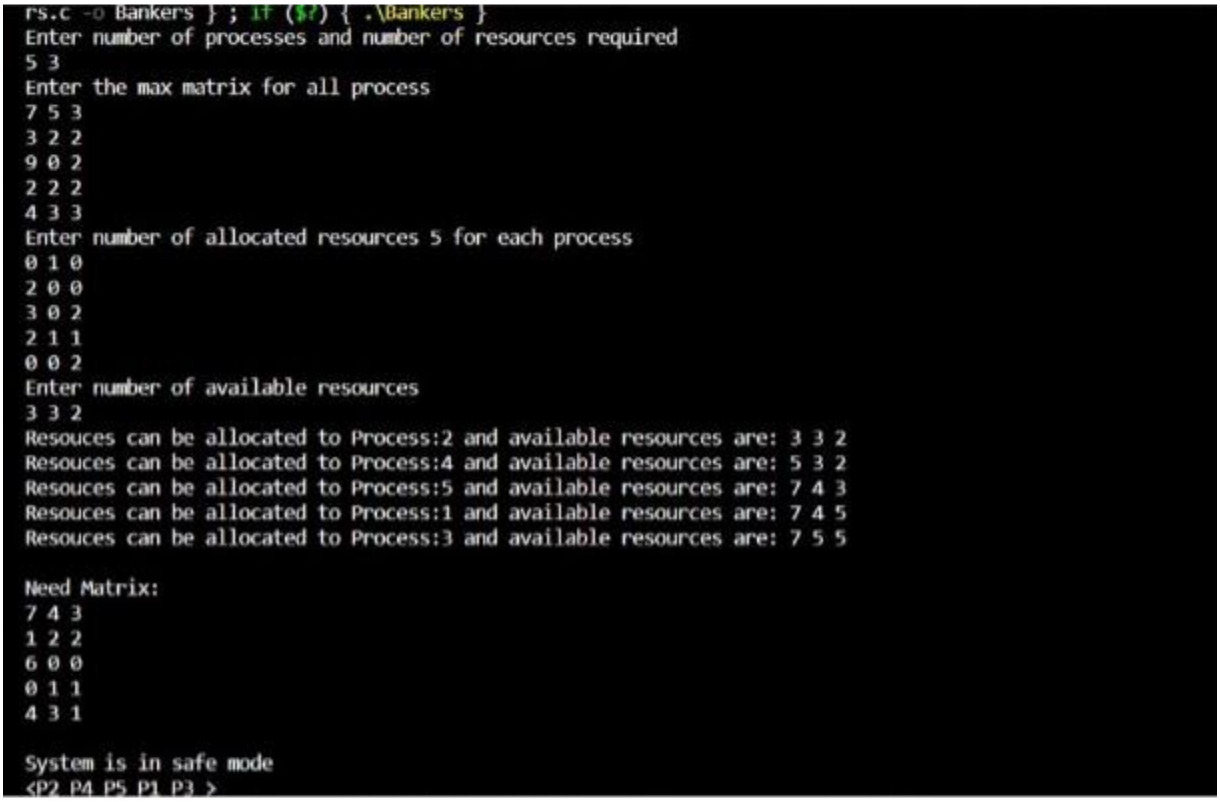
5. Bankers Algorithm

```
#include <stdio.h>
int main() {
    int n, m, i, j, k;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the number of resources: ");
    scanf("%d", &m);
    int allocation[n][m];
    int max[n][m];
    int available[m];
    int need[n][m];
    int finish[n], safeSeq[n], index = 0;
    printf("Enter the Allocation Matrix:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
        {
```

```
        scanf("%d", &allocation[i][j]);
    }
}
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 Available Resources:\n");
for (i = 0; i < m; i++)
{
    scanf("%d", &available[i]);
}
for (i = 0; i < n; i++)
{
    for (j = 0; j < m; j++)
    {
        need[i][j] = max[i][j] - allocation[i][j];
    }
}
for (i = 0; i < n; i++)
{
    finish[i] = 0;
}
for (k = 0; k < n; k++)
{
    for (i = 0; i < n; i++)
    {
        if (finish[i] == 0)
        {
            int flag = 1;
            for (j = 0; j < m; j++)
            {
                if (need[i][j] > available[j])
                {
                    flag = 0;
                    break;
                }
            }
            if (flag == 1)
            {
                safeSeq[index++] = i;
```

```c
                for (j = 0; j < m; j++)
                {
                    available[j] += allocation[i][j];
                }
                finish[i] = 1;
            }
        }
    }
}
int allFinished = 1;
for (i = 0; i < n; i++)
{
    if (finish[i] == 0)
    {
        allFinished = 0;
        break;
    }
}
if (allFinished)
{
    printf("Following is the SAFE Sequence:\n");
    for (i = 0; i < n - 1; i++)
    {
        printf("P%d -> ", safeSeq[i]);
    }
    printf("P%d\n", safeSeq[n - 1]);
}
else
{
    printf("The system is NOT in a safe state.\n");
}

return 0;
}
```

Output:



```
rs.c -o Bankers } ; if ($?) { .\Bankers }
Enter number of processes and number of resources required
5 3
Enter the max matrix for all process
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter number of allocated resources 5 for each process
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter number of available resources
3 3 2
Resouces can be allocated to Process:2 and available resources are: 3 3 2
Resouces can be allocated to Process:4 and available resources are: 5 3 2
Resouces can be allocated to Process:5 and available resources are: 7 4 3
Resouces can be allocated to Process:1 and available resources are: 7 4 5
Resouces can be allocated to Process:3 and available resources are: 7 5 5

Need Matrix:
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1

System is in safe mode
<P2 P4 P5 P1 P3 >
```

6. Deadlock Detection

```
#include <stdio.h>
static int mark[20];
int i, j, np, nr;
int main()
{
    int alloc[10][10], request[10][10], avail[10], r[10], w[10];
    printf("\nEnter the number of processes: ");
    scanf("%d", &np);
    printf("\nEnter the number of resources: ");
    scanf("%d", &nr);
    for (i = 0; i < nr; i++)
    {
        printf("Total amount of Resource R%d: ", i + 1);
        scanf("%d", &r[i]);
    }
    printf("\nEnter the Request Matrix:\n");
    for (i = 0; i < np; i++)
    {
        for (j = 0; j < nr; j++)
        {
```

```
        scanf("%d", &request[i][j]);
    }
}
printf("\nEnter the Allocation Matrix:\n");
for (i = 0; i < np; i++)
{
    for (j = 0; j < nr; j++)
    {
        scanf("%d", &alloc[i][j]);
    }
}
for (j = 0; j < nr; j++)
{
    avail[j] = r[j];
    for (i = 0; i < np; i++)
    {
        avail[j] -= alloc[i][j];
    }
}
for (i = 0; i < np; i++)
{
    int count = 0;
    for (j = 0; j < nr; j++)
    {
        if (alloc[i][j] == 0)
            count++;
        else
            break;
    }
    if (count == nr)
        mark[i] = 1;
}
for (j = 0; j < nr; j++)
    w[j] = avail[j];
for (i = 0; i < np; i++)
{
    int canBeProcessed = 0;
    if (mark[i] != 1)
    {
        for (j = 0; j < nr; j++)
        {
            if (request[i][j] <= w[j])
                canBeProcessed = 1;
            else {
                canBeProcessed = 0;
```

```
                break;
            }
        }
        if (canBeProcessed)
        {
            mark[i] = 1;
            for (j = 0; j < nr; j++)
                w[j] += alloc[i][j];
        }
    }
}
int deadlock = 0;
for (i = 0; i < np; i++)
{
    if (mark[i] != 1)
    {
        deadlock = 1;
        break;
    }
}

if (deadlock)
    printf("\nDeadlock detected.\n");
else
    printf("\nNo Deadlock possible.\n");

return 0;
}
```

Output:

```
Enter the number of processes: 3

Enter the number of resources: 2
Total amount of Resource R1: 12
Total amount of Resource R2: 16

Enter the Request Matrix:
1
2
7
3
9
5

Enter the Allocation Matrix:
1
5
2
7
3
8

Deadlock detected.
```

7. First fit, Best fit, Worst fit

```
 #include <stdio.h>
#define MAX 25

void firstFit(int b[], int nb, int f[], int nf);
void worstFit(int b[], int nb, int f[], int nf);
void bestFit(int b[], int nb, int f[], int nf);

int main() {
    int b[MAX], f[MAX], nb, nf;
    printf("Memory Management Schemes\n");
    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 (int i = 0; i < nb; i++)
    {
```

```
        printf("Block %d: ", i + 1);
        scanf("%d", &b[i]);
    }

    printf("\nEnter the size of the files:\n");
    for (int i = 0; i < nf; i++)
    {
        printf("File %d: ", i + 1);
        scanf("%d", &f[i]);
    }

    printf("\nMemory Management Scheme - First Fit");
    firstFit(b, nb, f, nf);

    printf("\n\nMemory Management Scheme - Worst Fit");
    worstFit(b, nb, f, nf);

    printf("\n\nMemory Management Scheme - Best Fit");
    bestFit(b, nb, f, nf);

    return 0;
}

void firstFit(int b[], int nb, int f[], int nf)
{
    int bf[MAX] = {0}, ff[MAX] = {0}, frag[MAX];

    for (int i = 0; i < nf; i++)
    {
        ff[i] = -1;
        for (int j = 0; j < nb; j++)
        {
            if (!bf[j] && b[j] >= f[i])
            {
                ff[i] = j;
                bf[j] = 1;
                frag[i] = b[j] - f[i];
                break;
            }
        }
    }

    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
    for (int i = 0; i < nf; i++)
    {
```

```
        if (ff[i] != -1)
            printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], ff[i] + 1, b[ff[i]], frag[i]);
        else
            printf("\n%d\t\t%d\t\tNot Allocated", i + 1, f[i]);
    }
}

void worstFit(int b[], int nb, int f[], int nf)
{
    int bf[MAX] = {0}, ff[MAX] = {0}, frag[MAX];

    for (int i = 0; i < nf; i++)
    {
        int worstIdx = -1;
        for (int j = 0; j < nb; j++)
        {
            if (!bf[j] && b[j] >= f[i])
            {
                if (worstIdx == -1 || b[j] - f[i] > b[worstIdx] - f[i])
                {
                    worstIdx = j;
                }
            }
        }
        ff[i] = worstIdx;
        if (worstIdx != -1)
        {
            bf[worstIdx] = 1;
            frag[i] = b[worstIdx] - f[i];
        }
    }

    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
    for (int i = 0; i < nf; i++)
    {
        if (ff[i] != -1)
            printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], ff[i] + 1, b[ff[i]], frag[i]);
        else
            printf("\n%d\t\t%d\t\tNot Allocated", i + 1, f[i]);
    }
}

void bestFit(int b[], int nb, int f[], int nf)
{
    int bf[MAX] = {0}, ff[MAX] = {0}, frag[MAX];
```

```
    for (int i = 0; i < nf; i++)
    {
        int bestIdx = -1;
        for (int j = 0; j < nb; j++)
        {
            if (!bf[j] && b[j] >= f[i])
            {
                if (bestIdx == -1 || b[j] - f[i] < b[bestIdx] - f[i])
                {
                    bestIdx = j;
                }
            }
        }
        ff[i] = bestIdx;
        if (bestIdx != -1)
        {
            bf[bestIdx] = 1;
            frag[i] = b[bestIdx] - f[i];
        }
    }

    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
    for (int i = 0; i < nf; i++)
    {
        if (ff[i] != -1)
            printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], ff[i] + 1, b[ff[i]], frag[i]);
        else
            printf("\n%d\t\t%d\t\tNot Allocated", i + 1, f[i]);
    }
}
```

Output:

```
Memory Management Schemes

Enter the number of blocks: 4
Enter the number of files: 2

Enter the size of the blocks:
Block 1: 34
Block 2: 12
Block 3: 22
Block 4: 25

Enter the size of the files:
File 1: 11
File 2: 15

Memory Management Scheme - First Fit
File_no:    File_size:  Block_no:   Block_size: Fragment
1       11      1       34      23
2       15      3       22      7

Memory Management Scheme - Worst Fit
File_no:    File_size:  Block_no:   Block_size: Fragment
1       11      1       34      23
2       15      4       25      10

Memory Management Scheme - Best Fit
File_no:    File_size:  Block_no:   Block_size: Fragment
1       11      2       12      1
2       15      3       22      7
```

8. LRU-Optimal-FIFO

```
#include <stdio.h>

int n, f, i, j, k;
int in[100];
int p[50];
int hit = 0;
int pgfaultcnt = 0;

void getData() {
    printf("\nEnter length of page reference sequence: ");
    scanf("%d", &n);
```

```
    printf("\nEnter the page reference sequence: ");
    for(i = 0; i < n; i++)
        scanf("%d", &in[i]);
    printf("\nEnter number of frames: ");
    scanf("%d", &f);
}

void initialize() {
    pgfaultcnt = 0;
    for(i = 0; i < f; i++)
        p[i] = 9999;
}

int isHit(int data) {
    hit = 0;
    for(j = 0; j < f; j++) {
        if(p[j] == data) {
            hit = 1;
            break;
        }
    }
    return hit;
}

void dispPages() {
    for (k = 0; k < f; k++) {
        if(p[k] != 9999)
            printf(" %d", p[k]);
    }
    printf("\n");
}

void dispPgFaultCnt() {
    printf("\nTotal number of page faults: %d\n", pgfaultcnt);
}

void fifo() {
    initialize();
    int index = 0;
    for(i = 0; i < n; i++) {
        printf("For %d :", in[i]);
        if(isHit(in[i]) == 0) {
            p[index] = in[i];
            index = (index + 1) % f;
            pgfaultcnt++;
```

```
            printf(" Page Fault ->");
            dispPages();
        } else {
            printf(" No page fault\n");
        }
    }
    dispPgFaultCnt();
}

void optimal() {
    initialize();
    int near[50];
    for(i = 0; i < n; i++) {
        printf("For %d :", in[i]);
        if(isHit(in[i]) == 0) {
            for(j = 0; j < f; j++) {
                int pg = p[j];
                int found = 0;
                for(k = i + 1; k < n; k++) {
                    if(pg == in[k]) {
                        near[j] = k;
                        found = 1;
                        break;
                    }
                }
                if(!found)
                    near[j] = 9999;
            }
            int max = -1, repindex = -1;
            for(j = 0; j < f; j++) {
                if(near[j] > max) {
                    max = near[j];
                    repindex = j;
                }
            }
            p[repindex] = in[i];
            pgfaultcnt++;
            printf(" Page Fault ->");
            dispPages();
        } else {
            printf(" No page fault\n");
        }
    }
    dispPgFaultCnt();
}
```

```
void lru() {
    initialize();
    int least[50];
    for(i = 0; i < n; i++) {
        printf("For %d :", in[i]);
        if(isHit(in[i]) == 0) {
            for(j = 0; j < f; j++) {
                int pg = p[j];
                int found = 0;
                for(k = i - 1; k >= 0; k--) {
                    if(pg == in[k]) {
                        least[j] = k;
                        found = 1;
                        break;
                    }
                }
                if(!found)
                    least[j] = -1;
            }
            int min = 9999, repindex = -1;
            for(j = 0; j < f; j++) {
                if(least[j] < min) {
                    min = least[j];
                    repindex = j;
                }
            }
            p[repindex] = in[i];
            pgfaultcnt++;
            printf(" Page Fault ->");
            dispPages();
        } else {
            printf(" No page fault\n");
        }
    }
    dispPgFaultCnt();
}

int main() {
    int choice;
    while(1) {
        printf("\nPage Replacement Algorithms\n");
        printf("1. Enter data\n");
        printf("2. FIFO\n");
        printf("3. Optimal\n");
```

```
        printf("4. LRU\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice) {
            case 1: getData(); break;
            case 2: fifo(); break;
            case 3: optimal(); break;
            case 4: lru(); break;
            case 5: return 0;
            default: printf("Invalid choice. Try again.\n");
        }
    }
}
```

Output:

```
Page Replacement Algorithms
1. Enter data
2. FIFO
3. Optimal
4. LRU
5. Exit
Enter your choice: 1

Enter length of page reference sequence: 4

Enter the page reference sequence: 3 6 1 8

Enter number of frames: 2

Page Replacement Algorithms
1. Enter data
2. FIFO
3. Optimal
4. LRU
5. Exit
Enter your choice: 2
For 3 : Page Fault -> 3
For 6 : Page Fault -> 3 6
For 1 : Page Fault -> 1 6
For 8 : Page Fault -> 1 8

Total number of page faults: 4
```

```
Page Replacement Algorithms
1. Enter data
2. FIFO
3. Optimal
4. LRU
5. Exit
Enter your choice: 3
For 3 : Page Fault -> 3
For 6 : Page Fault -> 6
For 1 : Page Fault -> 1
For 8 : Page Fault -> 8

Total number of page faults: 4

Page Replacement Algorithms
1. Enter data
2. FIFO
3. Optimal
4. LRU
5. Exit
Enter your choice: 4
For 3 : Page Fault -> 3
For 6 : Page Fault -> 3 6
For 1 : Page Fault -> 1 6
For 8 : Page Fault -> 1 8

Total number of page faults: 4
```