VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

VARSHA M GOWDA (1WA23CS033)

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 to 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 – 23CS4PCOPS" carried out by Student name (USN), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025- June 2025. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS - (23CS4PCOPS) work prescribed for the said degree.

Faculty Incharge Name
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

INDEX

NAME: Varsha. M. Gowda STD.: _____ SEC.: 4-G ROLL NO.: 033 SUB.: ~~UNIX SHELL PROGRAMMING~~ operating system.

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
		FCFS		
		SJF (Non-preemptive)		
		SRTF		
		Priority (Preemptive) ~~Non-Preemptive~~		
		Multilevel queue		
		Rate Monotonic		
		Round robin		
		priority		
		Dinning philosophy		
		producer comsumer		
		Deadline		
		Bankers		
		Deadlock		
		Memory allocation		
		Page riplacement.		

Course Outcomes

C01	Apply the different concepts and functionalities of Operating System
C02	Analyse various Operating system strategies and techniques
C03	Demonstrate the different functionalities of Operating System.
C04	Conduct practical experiments to implement the functionalities of Operating system.

FCFS

```
#include<stdio.h>

typedef struct{
    int id, arrival, burst, remaining, waiting,
turnaround, completion, started;
    }process;

Void swap(process *a, process *b){
    process temp = *a;
    *a = *b;
    *b = temp;
}

Void sortbyArrival (process p[], int n){
    for(int i=0; i<n-1; i++){
    for(int j=0; j<n-i-1; j++){
        if(p[j].arrival > p[j+1].arrival){
            swap(&p[j], &p[j+1]);
    }
    }
}
}
```

```
void fcfs (process p[], int n){
    sort by arrival (p, n);
    int time = 0;

    for(int i=0; i<n ; i++){
        if (time < p[i].arrival)
            time = p[i].arrival;

        p[i].copletion = time + p[i].burst;
        p[i].turnaround = p[i].completion - p[i].arrival;
        p[i].waiting = p[i].turnaround - p[i].burst;
        time = p[i].completion;
    }
}

void displayResult (process p[], int n, const char *title){
    printf("\n---%s---\n", title);
    printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");

    float totalWT = 0, totalTAT = 0;
    for(int i=0; i<n; i++){
    printf("P%d\t%d\t%d\t%d\t%d\t%d\n",
      p[i].id, p[i].arrival, p[i].burst, p[i].completion,
      p[i].turnaround, p[i].waiting);
    totalWT += p[i].waiting;
    totalTAT += p[i].turnaround;
}
```

```
printf("Average waiting time : %.2f \n", totalWT/n);
printf("Average turnaround time : %.2f \n", totalTAT/n);
}

int main(){
    int n;
    printf("Enter number of process :");
    scanf("%d", &n);
    process p[n], temp[n];

    Printf("Enter Arrival Time & Burst time: \n");
    for (int i=0; i<n; i++) {
        p[i].id = i+1;
        Scanf("%d %d", &p[i].arrival, &p[i].burst);
        p[i].remaining = p[i].burst;
        p[i].waiting = p[i].turnaround = p[i].
                completion = p[i].started = 0;
        temp[i] = p[i];
    }
    for (int i=0; i<n; i++) p[i] = temp[i];
        fcfs(p, n);
        displayResults(p, n, "First come First
                serve (FCFS)");
    return 0;
}
```

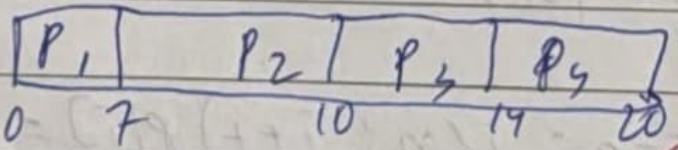
o/p: Enter the number of process: 4
Enter Arrival Time & Burst time:
0 7
0 3
0 4
0 6

FCFS

Process	AT	BT	CT	TAT	Wt
P_1	0	7	7	7	0
P_2	0	3	10	10	7
P_3	0	4	14	14	10
P_4	0	6	20	20	14

Average turnaround time: 12.75
Average waiting tim 7.75

gantt chat



6/3/25

2) SJF (Non preemptive)

```
void sjfNon(process p[], int n) {
    int completed = 0, time = 0;
    while (completed < n) {
        int min = -1, minburst = 100;
        for (int i = 0; i < n; i++) {
            if (p[i].arrival <= time && p[i].completion
                == 0) {
                if (p[i].burst < minburst || (p[i].burst == minburst
                    && p[i].arrival < p[min].arrival)) {
                    minburst = p[i].burst;
                    min = i;
                }
            }
        }
        if (min == -1) {
            time++;
            continue;
        }
        p[min].remaining = time - p[min].arrival;
        time += p[min].burst;
        p[min].completion = time;
        p[min].turnaround = p[min].completion
                            - p[min].arrival
```

p[min], waiting = p[min]. turnaround - p[min].burst
completed = 1;
}
}

Enter number of process: 4
Enter Arrival time & Burst time for each process
0 7
0 3
0 4
0 6
Shortest Job First (Non-preemptive

Process	AT	BT	CT	TAT	WT
P_1	0	7	20	20	13
P_2	0	3	3	3	0
P_3	0	4	7	7	3
P_4	0	6	13	13	7

Avg turnaround time = 10.75
Avg waiting time: 5.75

=> SJF (Preemptive) / SRTF

```
Void sjf-preemptive(Process p[], int n, float *avgTAT,
  float *avgWT) {
  int completed =0, time =0, minIdx, totalTAT=0,
  total WT=0;
  int iscompleted[n];
  for (int i =0; i<n; i++){
    Iscompleted[i] =0;
    p[i]. remaining =p[i]-burst;
  }
  While (completed <n){
      minIdx =-1;
      int minBurst = INT_MAX;
      for (int i =0; i<n; i++){
        If (!iscompleted[i] && p[i]. arrival <=
        time && p[i]. remaining < minBurst
        && p[:]. remaining >0){
          minBurst = p[i]. remaining
          minIdx =i;
    }
  }
If (minIdx == -1) {time++; continue;}
P[minIdx]. remaining --;
time++;
```

```
if (p[minIdx].remaining == 0){
    p[minIdx].completion = time;
    p[minIdx].turnaround = p[minIdx].completion
                          - p[minIdx].arrival;
    p[minIdx].waiting = p[minIdx].turnaround -
                        p[minIdx].burst;
    isCompleted[minIdx] = 1;
    totalTAT += p[minIdx].turnaround;
    totalWT += p[minIdx].waiting;
    completed++;
  }
 }
 *avgTAT = (float)totalTAT / n;
 *avgWT = (float)totalWT / n;
}

void display(struct process p[], int n, float
 avgTAT, float avgWT){
printf("\n PID Arrival Burst completion Turnaround
       waiting \n");
  for(int i=0; i<n; i++){
  printf("%3d %7d %6d %10d %10d %8d \n",
  p[i].id, p[i].arrival, p[i].burst, p[i].completion,
  p[i].turnaround, p[i].waiting);
}
```

```
printf("\n Average Turnaround Time : %.2f", avgTAT);
printf("\n Average Waiting Time : %.2f\n", avgWT);
}

int main(){
    int i;
    float avgTAT, avgWT;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process p[n];
    printf("Enter Arrival Time & Burst time
    for each process: \n");
     for (int i=0; i<n; i++){
        p[i].id = i+1;
        printf("P[%d]: ", i+1);
        scanf("%d %d", &p[i].arrival, &p[i].burst);
    }
```

```
Enter no of processes: 4
Enter Arrival time and Burst time for each
process:
P[1]:  0 7
P[2]:  8 3
P[3]:  3 2
P[4]:  5 6
```

SJB (Preemptive) scheduling

PID	AT	BT	CT	TAT	WT
1	0	7	9	9	2
2	8	3	12	4	1
3	3	2	5	2	0
4	5	6	18	13	7

Avg TAT : 7.00

Avg WT : 2.5

Priority

```
#include <stdio.h>
#define MAX 10
typedef struct {
    int pid, at, bt, pt, remaining_bt, ct, tat, wt,
        rt, is_completed, st;
} process;
void nonPreemptivePriority(process p[], int n){
    int time = 0, completed = 0;
    while (completed < n){
        int lowest_priority = 9999, selected = -1;
        for(int i = 0; i < n; i++){
            if (p[i].at <= time && !p[i].is_completed &&
                p[i].pt < lowest_priority){
                lowest_priority = p[i].pt;
                selected = i;
            }
        }
        if (selected == -1){
            time++;
            continue;
        }
        if (p[selected].rt == -1){
            p[selected].st = time;
            p[selected].rt = time - p[selected].at;
        }
```

```
        time += p[selected].bt;
        p[selected].ct = time;
        p[selected].tat = p[selected].ct - p[selected].at;
        p[selected].wt = p[selected].tat - p[selected].bt;
        p[selected].is_completed = 1;
        completed++;
    }
}
}

void preemptivePriority(Process p[], int n) {
    int time = 0, completed = 0;
    while(completed < n) {
        int lowest_priority = 9999, selected = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].remaining_bt > 0 &&
                p[i].pt < lowest_priority) {
                lowest_priority = p[i].pt;
                selected = i;
            }
        }
        if (p[selected].rt == -1) {
            p[selected].st = time;
            p[selected].rt = time - p[selected].at;
        }
        p[selected].remaining_bt--;
        time++;
        if (p[selected].remaining_bt == 0) {
            p[selected].ct = time;
            p[selected].tat = p[selected].ct - p[selected].at;
            p[selected].wt = p[selected].tat - p[selected].bt;
            completed++;
        }
    }
}

void displayProcesses(Process p[], int n) {
    float avg_tat = 0, avg_wt = 0, avg_rt = 0;
    printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
        p[i].pid, p[i].at, p[i].bt, p[i].pt, p[i].ct,
        p[i].tat, p[i].wt, p[i].rt);
        avg_tat += p[i].tat;
        avg_tat += p[i].wt;
        avg_rt += p[i].rt;
    }
    printf("\nAverage TAT: %.2f", avg_tat/n);
    printf("\nAverage WT: %.2f", avg_wt/n);
    printf("\nAverage RT: %.2f", avg_rt/n);
}
int main() {
    Process p[MAX];
    int n, choice;
    printf("Enter the number of processes:");
```

```
scanf("%d", &n);
for(int i=0; i<n; i++) {
    p[i].pid = i+1;
    printf("\nEnter Arrival Time, Burst Time, &
        Priority for process %d: \n", p[i].pid);
    printf("Arrival Time: ");
    scanf("%d", &p[i].at);
    printf("Burst Time: ");
    scanf("%d", &p[i].bt);
    printf("Priority(lower number mean higher
        priority): ");
    scanf("%d", &p[i].pt);
    p[i].remaining_bt = p[i].bt;
    p[i].is_completed = 0;
    p[i].rt = -1;
}
while(1) {
    printf("\n Priority Scheduling Menu: \n");
    printf("1. Non-preemptive priority Scheduling\n");
    printf("2. preemptive priority Scheduling\n");
    printf("3. Exit \n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch(choice) {
        case 1:
            nonPreemptive Priority(p, n);
            printf("Non-Preemptive Scheduling Completed");
            displayProcesses(p, n);
            break;
        case 2:
            preemptive Priority(p, n);
            printf("Preemptive Scheduling Completed");
            display processes(p, n);
            break;
        case 3:
            printf("Exiting...\n");
            return 0;
        default:
            printf("Invalid Choice! try again.\n");
    }
}
return 0;
}
```

O/P: priority (lower number means higher priority): 2
Enter arrival time, Burst time, and priority
for process 3:
Arrival time: 2
Burst time: 2
Prority (lower number mean higher priority):

Enter Arrival Time, Burst time, & priority for process 1: 4 4 3

process 1: 0 5 4
process 2: 2 4 2

priority Scheduling Menu:
1. Non-Preemptive priority Scheduling
2. Preemptive priority scheduling
3. Exit
Enter your choice: 1
Non-preemptive Scheduling completed:

PID	AT	BT	Priority	CT	TAT	WT	RT
1	0	5	4	5	5	0	0
2	2	4	2	9	7	3	3
3	2	2	6	15	13	11	11
4	4	4	3	13	9	5	5

Average TAT: 8.5
Average WT: 4.75
Average RT: 4.75

priority Scheduling menu:
1. Non-Preemptive priority scheduling
2. Preemptive priority scheduling
3. Exit
Enter your choice: 2
preemptive scheduling completed!

PID	AT	BT	Priority	CT	TAT	WT	RT
1	0	5	4	13	13	8	0
2	2	4	2	6	4	0	3
3	2	2	6	15	13	11	11
4	4	4	3	10	6	2	5

Average TAT: 9.00
Average WT: 5.25
Average RT: 4.75

Round Robin

```
#include <stio.h>
#define MAX 100

void roundrobin (int n, int at[], int bt[], int
                                    quant){
  int ct[n], tat[n], wt[n], rem_bt[n];
  int queue[MAX], front=0, rear=0;
  int time=0, completed=0, visited[n];
   for (int i=0; i<n; i++){
     rem_bt[i] = bt[i];
      visited[i] = 0;
  }
  queue[rear++] = 0;
  visited[0] = 1;
 while (completed <n) {
   int index = queue[front++];
 if (rem_bt[index] > quant) {
      time += quant;
      rem_bt[index] -= quant;
  } else{
     time += rem_bt[index];
    rem_bt[index]=0;
     ct[index] =time;
     completed ++;
}
```

```
for (int i=0; i<n; i++){
    if (at[i] <= time && rem_bt[i] >0 &&
     !visited[i]){
    queue[rear++]=i;
    visited[i] =1;
  }
}
if (rem_bt[index] >0){
     queue[rear++] = index;
}

if (front == rear){
   for (int i=0; i<n; i++){
       if (rem_bt[i]>0){
          queue[rear++] = i;
          visited[i]=1;
          break;
       }
   }
}
float total_tat =0; total_wt =0;
printf("P# \tAT\tBT\tCT\tTAT\tWT\n");
 for (int i=0; i<n; i++){
```

```
tat[i] = ct[i] - at[i];
wt[i] = tat[i] - bt[i];
total_tat += tat[i];
total_wt += wt[i];
printf("%d\t%d\t%d\t%d\t%d\t%d\n", i+1,
      at[i], bt[i], ct[i], tat[i], wt[i]);
}
printf("Average TAT : %.2f\n", total_tat/n);
printf("Average WT : %.2f\n", total_wt/n);
}
int main(){
  int n, quant;
  printf("Enter number of processes : ");
  scanf("%d", &n);

 int at[n], bt[n];
  for (int i=0; i<n; i++){
     printf("Enter AT and BT for process %d:", i+1);
     scanf("%d %d", &at[i], &bt[i]);
  }
  printf("Enter time quantum:");
  scanf("%d", &quant);
 roundrobin(n, at, bt, quant);
  return 0;
}
```

Enter number of processes: 5
Enter AT and BT for process 1: 0 8
Enter AT and BT for process 2: 5 2
Enter AT and BT for process 3: 1 7
Enter AT & BT for process 4: 6 3
Enter AT & BT for process 5: 8 5
Enter time quantum: 3

P#	AT	BT	CT	TAT	WT
1	0	8	22	22	14
2	5	2	11	6	4
3	1	7	23	22	15
4	6	3	14	8	5
5	8	5	25	17	12

Avg TAT = 15.00
Avg WT = 10.00

multi level queue

using RR & FCFS

```
#include <stdio.h>
#define MAX_Processes 10
#define Time_QUANTUM 2

typedef struct{
    int burst_time, arrival_time, queue_type
    WT, TAT, RT, remaining_time;
} process;

void roundrobin (process P[], int n, int
                 time_quantum, int *time){
    int done, i;
    do{
        done = 1;
    for (i=0; i<n; i++){
        if (P[i].remaning_time>0){
            done = 0;
        if (p[i].remaining_time > time_quantum)
        {
    *time += time_quantum;
    processes[i].remaining_time -= time_quant
    -um } do{
    *time += p[i].remaining_time;
```

```
p[i].waiting_time = *time-p[i].arrival_time -
                    p[i].burst_time;
p[i].turnaround_time = *time-p[i].arrival_time;
p[i].response_time = p[i].waiting_time;
p[i].remaining_time = 0;
}
}
}
} while(!done);
}
void fcfs(process p[], int n, int *time){
for(int i=0; i<n; i++){
if(*time < p[i].arrival_time){
*time = p[i].arrival_time;
}
p[i].waiting_time = *time - p[i].arrival_time;
p[i].turnaround_time = p[i].waiting_time +
                       p[i].burst_time;
p[i].response_time = p[i].waiting_time;
*time += p[i].burst_time;
}
}
```

o/p

```
Enter number of processes: 4
Enter Burst time, Arrival time & Queue no: 2 0 1
Enter Burst time, Arrival time & Queue no: 2 0 2
Enter Burst time, Arrival time & Queue no: 5 0 1
Enter Burst time, Arrival time & Queue no: 3 0 2
Queue 1 is system process
Queue 2 is user process
Process  WT  TAT  RT
1        0   2    0
2        2   7    2
3        7   8    7
4        8   11   8
Avg WT: 4.25
Avg TAT: 1.00
Avg RT: 4.25
Throughput: 0.36
Process returned 11 (0x11)  Execution time: 11.55s
```

=> WAP to simulate Real time CPU Scheduling

a) Rate-Monotonic

```
#include <stdio.h>
#define MAX_PROCESS 10

typedef struct{
    int id;
    int burst_time;
    int period;
    int Remaining_time;
    int next_deadline;
} process;
void sort_by_period (process processes[],
                     int n) {
    for(int i=0; i<n-1; i++){
        for(int j=0; j<n-i-1; j++){
            if(processes[j].period > processes[j+1].period){
                process temp = processes[j];
                processes[j] = processes[j+1];
                processes[j+1] = temp;
            }
        }
    }
}
```

```
int gcd(int a, int b){
    return b==0? a : gcd(b, a%b);
}
int lcm(int a, int b) {
    return (a*b)/gcd(a,b);
}
int calculate_lcm(process processes[], int n){
    int result = processes[0].period;
    for(int i=1; i<n; i++) {
        result = lcm(result, processes[i].period);
    }
    return result;
}
double utilization_factor(process processes[], int n){
    double sum = 0;
    for(int i=0; i<n; i++){
        sum += (double) processes[i].burst_time /
               processes[i].period;
    }
    return sum;
}
double rms_threshold(int n) {
    return n*(pow(2.0, 1.0/n) - 1);
}
```

```
void rate_monotonic_scheduling(process processes[],
                               int n){
int lcm_period = calculate_lcm(processes, n);
printf("LCM = %d\n\n", lcm_period);
printf("Rate Monotonic Scheduling:\n");
printf("PID Burst Period\n");
 for (int i=0; i<n; i++){
    printf("%d %d %d\n", processes[i].id,
      processes[i].burst_time, processes[i].
      period);
 }
double utilization = utilization_factor(processes);
double threshold = rms_threshold(n);
printf("%.6f <= %.6f => %s\n", utilization,
    threshold, (utilization <= threshold)?
    "true": "false");
if (utilization > threshold){
   printf("\n system may not be schedulable
        \n");
   return;
}

int timeline =0, Executed=0;
while (timeline < lcm_period) {
   int selected = -1;
```

```
for (int i=0; i<n; i++){
   if (timeline % processes[i].period ==0){
   processes[i].remaining_time = processes[i].burst_time;
   }
   if (processes[i].remaining_time >0) {
        selected = i;
        break;
   }
}
if (selected != -1){
   printf("Time %d: Process %d is running\n",
        timeline, processes[selected].id);
   processes[selected].remaining_time--;
   Executed++;
}else{
   printf("Time %d: CPU is idle\n", timeline);
}
 timeline++;
}
}

int main(){
   int n;
   process processes[MAX_PROCESSES];
   printf("Enter the number of processes:");
   scanf("%d", &n);
```

```
Printf(" Enter the CPU burst times :\n");
  for(int i=0 ; i<n ; i++){
     processes[i].id = i+1;
     Scanf("%d", &processes[i].burst_time);
     processes[i].remaining_time = processes[i].burst_time;
  }
printf("Enter the time periods :\n");
  for (int i=0; i<n; i++) {
     Scanf("%d", &processes[i].period);
  }
  sort_by_period(processes, n);
  rate_monotonic_scheduling(processes, n);
  return 0;
}
```

```
Op   Enter the number of processes : 3
     Enter the cpu burst time: 3  6  8
     Enter the time periods : 3  4  6
     LCM = 60
     rate monotone scheduling:
     PID     Burst     Period
     1         3         3
     2         6         4
     3         8         5
     4.100 <= 0.779763 => false
        System may not be schedulable!
```

Producer consumer

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 3;
int item = 0;
int wait (int s);
int signal (int s);
void producer ();
void consumer ();
int main (){
    int choice;
    printf ("producer-consumer problem
            simulation \n");
  while (1) {
printf("\n 1. Produce \n 2. consume \n 3. Exit\n");
printf("Enter your choice:");
scanf("%d", &choice);
switch (choice) {
  case 1:
    if ((mutex == 1) && (empty != 0)){
      producer ();
  } else{
    printf (" Buffer is full or mutex is
    locked. Cannot produce. \n");
  }
    break;
  case 2;
    if ((mutex == 1) && (full != 0)) {
      consumer ();
    } else{
      printf("Buffer is empty or mutex is
            locked. Cannot consume. \n");
      }
      break;
    case 3;
      exit (0);
      default;
        printf("Invalid choice. Try again \n");
    }
  }
  return 0;
}
int wait (int s){
    return --s;
  }
int signal (int s) {
    return ++s;
  }
void producer () {
  mutex = wait (mutex);
```

```
    empty = wait(empty);
    full = signal(full);
    item++;
    printf("Produced item %d\n", item);
    mutex = signal(mutex);
}
void consumer(){
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
printf("Consumed item %d\n", item);
    item--;
mutex = signal(mutex);
}
```

o/p:- Producer consumer problem simulation

```
1. Produce
2. Consume
3. Exit
Enter your choice: 1
Produced item: 1

1. Produce
2. Consume
3. Exit
Enter your choice = 1
Produced item: 2

1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumed item 2

1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumed item: 1

1. Produce
2. Consume
3. Exit
Enter your choice: 2
Buffer is empty or mutex is locked. Cannot
Consume

1. Produce
2. Consume
3. Exit
Enter your choice: 3
```

Dinning philosophy

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#define N 5
#define thinking 2
#define hungry 1
#define eating 0
#define Left (phnum +4) % N
#define Right (phnum +1) % N
int state[N];
int phil[N] = {0,1,2,3,4};
sem_t mutex;
sem_t s[N];
void test (int phnum);
void take_fork (int phnum);
void put_fork (int phnum);
void * philosopher (void* num);
int main (){
    int i;
    pthread_t thread_id [N];
    Sem_init (&mutex, 0, 1);
    for (i=0; i<N; i++);
        Seme_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;
}
void test (int phnum){
    if (state[phnum] == HUNGRY &&
        state[LEFT] != EATING &&
        State[RIGHT] != EATING) {
        state[phnum] = EATING;
        sleep (2);
    printf ("philosopher %d take fork %d
        and %d \n", Phnum +1, LEFT +1,
                Phnum +1);
    printf ("philosopher %d is eating \n", phnum+1);
    sem_post (&s[phnum]);
}
}
void take_fork (int phnum){
    sem_wait (&mutex);
    State[phnum] = HUNGRY;
```

```
printf("Philosopher %d is Hungry\n", phnum+1);
test(phnum);
sem_post(&mutex);
sem_wait(&S[phnum]);
sleep(1);
}
void put_fork(int phnum){
  sem_wait(&mutex);
  state[phnum] = thinking;
printf("Philosopher %d putting fork %d &
  %d down\n", phnum+1, left+1, phnum+1);
printf("Philosopher %d is thinking\n", phnum+1);
test(LEFT);
test(RIGHT);
sem_post(&mutex);
}
void* philosopher(void *num) {
    int *i = (int*) num;
    while(1) {
        sleep(1);
        take_fork(*i);
        sleep(1);
        put_fork(*i);
    }
}
```

```
philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
philosopher 5 is thinking
philosopher 5 is Hungry
philosopher 4 is Hungry
philosopher 3 is Hungry
philosopher 2 is Hungry
philosopher 1 is Hungry.
philosopher 1 takes forks 5 & 1
philosopher 1 is eating
philosopher 1 putting forks 5 and 1 down
philosopher 1 is thinking
philosopher 5 takes fork 4 and 5
philosopher 5 is eating
philosopher 2 takes fork 1 & 2
philosopher 2 is eating
philosopher 5 putting for 4 and 5 down
philosopher 5 is thinking
```

Deadline

```
#include <stdio.h>
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
int lcm(int a, int b) {
    return (a * b) / gcd(a, b);
}
struct process {
    int id, burst_time, deadline, period;
};
void earliest_deadline_first(struct process p[],
int n, int time_limit) {
    int time = 0;
    printf("Earliest Deadline Scheduling :\n");
    printf("PID\t Burst\t Deadline\t Period\n");
    for (int i=0; i<n; i++) {
        int earliest = -1;
        for (int i=0; i<n; i++) {
            if (p[i].burst_time > 0) {
                if (earliest == -1 || p[i].deadline <
p[earliest].deadline) {
                    earliest = i;
                }
            }
        }
        if (earliest == -1) break;
        printf("%d", &n);
        scanf p[earliest].burst_time--;
        time++;
    }
}
int main() {
    int n;
    printf("Enter the number of processes:");
    scanf("%d", &n);
    struct process processes[n];
    printf("Enter the CPU burst times:\n");
    for (int i=0; i<n; i++) {
        scanf("%d", &processes[i].burst_time);
        processes[i].id = i+1;
    }
printf("Enter the deadline:\n");
    for (int i=0; i<n; i++) {
        scanf("%d", &processes[i].deadline);
    }
```

```
printf("Enter the time periods:\n");
for(int i=0; i<n; i++){
    scanf("%d", &processes[i].period);
}

int hyperperiod = processes[0].period;
for(int i=1; i<n; i++){
    hyperperiod = lcm(hyperperiod,
                      processes[i].period);
}

printf("\nSystem will Execute for hyperperiod
(LCM of periods): %d ms\n", hyperiod);
earliest_deadline_first(processes, n, hyperperiod)
return 0;
}
```

```
Enter the number of processes: 3
Enter the cpu burst times: 2 3 4
Enter the deadlines: 1 2 3
Enter the time periods: 1 2 3
system will Execute for hyperperiod(LCM of
periods): 6 ms
Earliest deadline scheduling:
```

PID	Burst	deadline	Period
1	2	1	1
2	3	2	2
3	4	3	3

```
Scheduling occurs for 6ms
0ms: Task 1 is running
1ms: Task 1 is running
2ms: Task 2 is running
3ms: Task 2 is running
4ms: Task 2 is running
5ms: Task 3 is running
```

Bankers

```
#include <stdio.h>
#define MAX 10
#define Resource_types 3

Void calculateNeed ( int Need[MAX][RESOURCE_Types],
int max[MAX] Resource_types], int allocation[MAX]
[Resource_types], int numProcesses , int numResources){
  for(int i=0; i< numProcesses ; i++ ){
     for(int j=0; j<numResources ; j++ ){
       need[i][j] = max[i][j] - allocation[i][j];
     }
  }
}

int isSafeState (int processes , int resources , int
available[ ], int max[][Resource_types], int
allot[ ][Resource_types], int safeSeq[ ]){
   int finish[processes], work [resources];
  int count =0;
  for (int i=0; i< processes ; i++){
     finish[i] =0;
}
 for(int i= 0; i<resources ; i++ ){
     Work [i] = available [i];
 }
 While (count < processes ){
```

```
  int found =0;
  for (int p=0; p< processes; p++){
    if (finish[p] ==0 ){
    int canAllocate = 1;
    for(int r=0; r< resources ; r++){
     if (max[p][r] - allot[p][r] > work[r]){
       canAllocate =0;
       break;
     }
   }
  if (canAllocate){
    for (int r=0; r< resources; r++){
      work[r] += allot[p][r];
    }
    safeSeq[count ++] = p;
     finish[p] = 1;
     found = 1;
   }
  }
}

 If (! found){
    return 0;
  }
}
return 1;
}
```

```
int main(){
  int numProcesses, numResources;
  printf("Enter the number of processes:");
  Scanf("%d", &numprocesses);
  printf("Enter the number of resources:");
  scanf("%d", &numResources);

  int max[numProcesses][numResources], allocatio
  [numProcesses][numResources], available[numResource
  int need[numProcesses][numResources];
  int safeseq[numProcesses];

  Printf("\nEnter the maximum resources
  required for each process:\n");
    for(int i=0; i<numProcesses; i++){
      printf("Process P%d:", i);
      for(int j=0; j<numResources; j++){
      scanf("%d", &max[i][j]);
    }
  }

  printf("\nEnter the resources currently
  allocated to each process:\n");
    for(int i=0; i<numProcesses; i++){
      scanf("%d", &allocation[i][j]);
    }
  }
```

```
  printf("\nEnter the available resources:\n");
  for(int i=0; i<numResources; i++){
    scanf("%d", &available[i]);
  }
  calculateNeed(need, max, allocation, numProcesses,
  numResources);
  if(isSafeState(numProcesses, numResources,
  available, max, allocations, safeSeq)){
    printf("\nThe system is in a safe state.\n");
    printf("Safe sequence:");
    for(int i=0; i<numProcesses; i++){
    printf("P%d", SafeSeq[i]);
    }
    printf("\n");
  }else{
    printf("\nThe system is not in a safe state.\n");
  }
  return 0;
}
```

O/P:-

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the maximum resources required for
each process:
Process P0: 7 5 3
```

process P1 : 3 2 2
—11— P2 : 9 0 2
—11— P3 : 2 2 2
—11— P4 : 4 3 3

Enter the available resources :
3 3 2

The system is in a safe state.
safe sequence : P1 P3 P4 P0 P2

Deadlock.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
#define Resource_type 3

int isCyclic (int graph[MAX][MAX], int numProcesses){
    int visited[MAX] = {0};
    int recStack[MAX] = {0};
    int dfs(int graph[MAX][MAX], int process, int
    visited[MAX], int recStack[MAX]){
      if (recStack[process])
        return 1;
      if (visited[process])
        return 1;
      if (visited[process])
        return 0;
      visited[process] = 1;
      recStack[process] = 1;
      for (int i = 0; i<numProcesses; i++){
        if (graph[process][i] && dfs(graph,i, visited,
          recStack))
          return 1;
    }
    recStack[process] = 0;
    return 0;
}
```

```
for (int i=0; i< numprocesses; i++){
    if (!visited[i]){
        If (dfs (graph, i, visited, recstack))
        return 1;
    }
}
return 0;
}
int detectDeadlock (int numProcesses, int
allocation[MAX][Resource-type], int need[MAX]
[resource-type], int numResources){
    int graph[MAX][MAX] = {0};
for (int i=0; i<numProcesses; i++){
    for (int j= 0; j<num Processes ; j++){
        if(i!=j){
            int canAllocate = 1;
        for(int r=0; r<numResources; r++){
            if(need[i][r] > allocation[j][r]){
                canAllocate=0;
                break;
            }
        }
        if (canAllocate){
            graph[i][j]=1;
        }
    }
}
```

```
if (isCyclic (graph, numProcesses)){
    printf ("\n Deadlock detected in the system.\n");
    return 1;
}
Printf ("\n No deadlock detected in the system\n");
return 0;
}
void calculateNeed (int need[MAX][Resource-type],
    int max[max][Resource-types], int allocation
    [MAX][Resource-type], int numProcesses, int numResources){
    for (int i=0; i<numProcesses; i++){
        for (int j=0; j<numResources; j++){
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}
int main(){
    int numprocesses, numResources;
    printf("Enter the number of processes: ");
    Scanf("%d", &numProcesses);
    printf("Enter the number of resources:");
    &Scanf("%d", &numResources);
int max[numProcesses][numResources], allocation
[numProcesses][numResources], available[numResources]
int need[numProcesses][numResources];
```

```
printf("\nEnter the maximum resources required
    for each process: \n");
for(int i=0; i<numProcesses; i++){
  printf("Process Pid:", i);
  for(int j=0; j<numResources; j++){
    scanf("%d", &max[i][j]);
  }
}
printf("\nEnter the resources currently allocated
    to each process: \n");
for(int i=0; i<numProcesses; i++){
  printf("Process Pid: %d", i);
  for(int j=0; j<numResources; j++){
    scanf("%d", &allocation[i][j]);
  }
}

Printf("\nEnter the available resources: \n");
for(int i=0; i<numResources; i++){
  scanf("%d", &available[i]);
}
CalculateNeed(need, max, allocation, numProcesses,
    numResources);
detectDeadlock(numProcesses, allocation, need,
    numResources);
return 0;
}
```

o/p:

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the maximum resources required for each
 process:
process P0: 7 5 3
-11- P1: 3 2 2
-11- P2: 9 0 2
-11- P3: 2 2 2
-11- P4: 4 3 3

Enter the resource currently allocated to each
 process:
Process P0: 0 1 0
-11- P1: 2 0 0
-11- P2: 3 0 2
-11- P3: 2 1 1
-11- P4: 0 0 2
Enter the available resources:
 3 3 2
No deadlock detected.
```

Memory allocation

```
#include <stdio.h>
struct Block{
    int size;
    int allocated;
};
struct file{
    int size;
    int block_no;
};
void reset Blocks (struct Block blocks[], int n){
  for (int i=0; i<n ; i++){
      blocks[i].allocated =0;
  }
}
Void first fit(struct Block blocks[], int n_blocks,
  Struct file files[], int n_files){
  printf ("\n\t memory Management Scheme -
        First Fit \n");
  Print f(" File no:\t File_size\t Block_no:\t Block_
        Size : \n");
for (int i=0; i<n_file ; i++){
    files[i]. block_no = -1;
  for (int j=0; j<n_blocks ;j++){
  if (!blocks[i]. allocated && blocks[j].size >=
    file[i]. size){
```

```
files[i]. block_no = j+1;
blocks[j]. allocated =1;
Printf ("%d \t\t %d \t\t %d \t\t %d\n", i+1,
        files[i]. size, j+1, blocks[j]. size);
  break;
  }
}
if files [i]. blockno ==-1){
  printf("%d \t\t %d \t\t _ \t\t _ \n", i+1,
        files[i]. size);
  }
}
Void best fit(struct block blocks[], int n_blocks,
    struct file files[], int n_files){
  printf ("\n Memory Mang ment scheme -Best fit \n");
  printf (" File no: \t File_size \t Block_no: \t Block_size\n");
for (int i=0; i<n_files; i++){
    int best Idx = -1;
  for (int j =0; j<n_blocks; j++){
  if(!blocks[j]. allocated && blocks[j].size >=
    files[i].size){
    if (best Idx == -1 || blocks[j]. size < blocks
      [best Idx]. size){
      best Idx = j;
}}}
```

```
if (bestIdx != -1){
    blocks[bestIdx].allocated = 1;
    file[i].block_no = bestIdx + 1;
    Printf("%d \t %d \t %d \t %d \n", i+1, file[i].size,
    bestIdx+1, blocks[bestIdx].size);
} else{
    printf("%d \t %d \t %d \t %d \n", i+1, file[i].size);
}
}

void worstfit(struct Block Blocks[], int n_blocks,
    struct file files[], int n_files){
printf("\n\t Memory Managment scheme - Worstfit");
printf("File no \t filesize \t Block no \t Block size:");
for(int i=0; i<n_files; i++){
    int worstIdx = -1;
for(int j=0; j<n_blocks; j++){
    if(!blocks[j].allocated && blocks[j].size >= file[i].size){
        if(worstIdx == -1 || block[j].size > blocks[worstIdx].size){
            worstIdx = j;
        }
    }
}
if(worstIdx != -1){
    blocks[worstIdx].allocated = 1;
    file[i].block_no = worstIdx + 1;
}
printf("%d \t %d \t %d \t %d \n", i+1, file[i].size);
}
```

```
int main(){
    int n_blocks, n_files, choice;
    printf("Memory Management Scheme \n");
    printf("Enter the number of blocks : ");
    scanf("%d", &n_blocks);
    printf("Enter the number of files : ");
    scanf("%d", &n_files);
    struct Block blocks[n_blocks];
    struct file files[n_files];
    printf("\n Enter the size of the blocks : \n");
    for(int i=0; i<n_blocks; i++){
        printf("Block %d :", i+1);
        scanf("%d", &blocks[i].size);
        blocks[i].allocated = 0;
    }
do{
    printf("\n 1. First Fit \n 2. Best fit \n 3.
        worst fit \n 4. Exit \n");
    printf("Enter your choice : ");
    scanf("%d", &choice);
    resetblocks(blocks, n_blocks);
    switch(choice){
        case 1:
            firstFit(blocks, n_blocks, files, n_files);
            break;
```

```
case 2:
    bestfit(block, n_blocks, file, n_file);
    break;
case 3
    Worstfit(blocks, n_blocks, files, n_files);
    break;
case 4:
    printf("\n Exiting ...\n");
    break;
default:
    printf("Invalid choice.\n");
}
} while (choice != 4);
return 0;
}
```

dp

```
Memory mangment scheme
Enter the no of blocks: 5
Enter the number of file: 4
Enter the size of the blocks:
Block 1: 100
Block 2: 500
Block 3: 200
-11- 4 : 300
-11- 5 = 600
Enter the size of the files:
file 1: 212
-11-2 : 417
-11-3 : 112
-11-4 : 426
1. First Fit
2. Best fit
3. Worst fit
4. Exit
Enter your choice: 1
```

First fit

File no:	Filesize	Block no:	Block size:
1	212	2	500
2	417	5	600
3	112	3	200
4	426	-	-

Enter the choice 2

Best fit

File no:	file size:	Block no:	Block size:
1	212	4	300
2	417	2	500
3	112	3	200
4	426	5	600

Enter your choice -3

worst fit

File no	file size	Block no	Block size
1	212	5	600
2	417	2	500
3	112	4	300
4	426	-	-

Enter your choice -4

Exiting ...

Page replacement-

```
#include <stdio.h>
#include <stdlib.h>
int findoptimal(int page[], int n, int frame[],
    int framesize, int index){
  int farthest = index, pos = -1;
  for(int i=0; i<framesize; i++){
      int j;
   for(j = index; j<n; j++){
     if(frame[i] == pages[j]){
          if(j>farthest){
              farthest = j;
          }   pos = i;
      break;
     }
   }
   if(j==n) return i;
  }
  return (pos == -1) ? 0 : pos;
 }
void fifo(int pages[], int n, int framesize){
    int frame[FramesIze];
     int front = 0, pagefaults = 0;
      int count=0;
  printf("\nFIFO Page replacement:\n");
```

```
for (int i=0; i<framesize; i++) frame[i]=-1;
for (int i=0; i<n; i++){
    int found=0;
  for(int j=0; j<frame size; j++){
        if (frame[j] == page[i]){
            found=1;
        break;
    }
  }
printf("Frames: ");
  for (int j=0; i<framesize; j++){
    if (frame[j] = -1)
        printf("%d", frame[j]);
    else
        printf("-");
  }
}

printf("Total page faults(FIFO): %d\n",
  Page faults);

void lru(int page[], int n, int frames){
  int frame[framesize], count[framesize];
  int time=0; pagefaults=0;
printf("\nLRU Page Replacement:\n");
```

```
for (int i=0; i<framesize; i++){
        frame[i]=-1;
        count[i]=0;
}
for (int i=0; i<n; i++){
        int found=0, pos=0, min=time;
  for (int j=0; j<framesize; j++){
    if (frame[j] == page[i]){
        found=1;
        count[j] = time++;
        break;
    }
  }
printf("Frames: ");
for (int j=0; j<frame size; j++){
  if (frame[j] != -1)
        printf("%d", frame[j]);
    else
        printf("-");
  }
}
printf("Total page faults(LRU): %d\n",
        Page faults);
}
```

```
void optimal(int pages[], int n, int frameSize){
    int frame[frameSize];
    int pageFaults=0;
    printf("optimal Page replacement\n");
  for(int i=0; i<frameSize; i++)frame[i]=-1;
for(int i=0; i<n; i++){
    int found=0;
   for(int j=0; j<frameSize; j++){
      if(frame[j]==pages[i]){
        found=1;
        break;
      }
   }
if(!found){
  int pos=-1;
  for(int j=0; j<frameSize; j++){
      if(frame[j]==-1){
          pos=j;
        break;
  }}
}
if(pos==-1){
    pos=findoptimal(pages, n, frame,
    frameSize, i+1);
}
```

```
frame[pos]=pages[i];
    pageFaults++;
}
Printf("Frames:");
 for(int j=0; j<frameSize; j++){
   if(frame[j]!=-1)
       printf("%d", frame[j]);
   else
       Printf("-");
 }
 }
 Printf("Total page faults(optimal): %d\n",
   pageFaults);
}
int main(){
 int n, frameSize, choice;

 int pages[n];
 printf("Enter page reference string:");
 do{
   Printf("\n --- Page Replacement Algorithm ---\n");
   Printf("1. FIFO\n2. LRU\n3. optimal\n4.
          exit\n");
   Printf("Enter your choice:");
```

```
switch(choice){
    case 1:
        fifo(page, n, framesize);
    case 3
    return 0;
}
```

O/P:

```
Enter the size of the page:
7
Enter the page strings:
1 3 0 3 5 6 3
Enter the no of page frames:
3
FIFO Page faults: 6, Page Hits: 1
optimal  -11-  : 5  -11-  : 2
LRU      -11-  : 7  -11-  : 0
```