VISVESVARAYA TECHNOLOGICAL UNIVERSITY
"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

C Mohan Babu(1BM23CS402)

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

Apr-2024 to Aug-2024

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled "OPERATING SYSTEMS – 23CS4PCOPS" carried C Mohan Babu**(1BM23CS402),** who is 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. 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.

Basavaraj Jakkali   **Dr. Jyothi S Nayak**

Assistant Professor   Professor and Head

Department of CSE   Department of CSE

BMSCE, Bengaluru   BMSCE, Bengaluru

Index Sheet

Program -1

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

FCFS

```c
#include <stdio.h>

struct Process {
    int pid;      // Process ID

    int burst_time; // Burst time

    int arrival_time; // Arrival time

    int waiting_time; // Waiting time

    int turnaround_time; // Turnaround time
};

void findWaitingTime(struct Process proc[], int n) {

    int service_time[n];

    service_time[0] = proc[0].arrival_time;

    proc[0].waiting_time = 0;


    for (int i = 1; i < n; i++) {

        service_time[i] = service_time[i-1] + proc[i-1].burst_time;

        proc[i].waiting_time = service_time[i] - proc[i].arrival_time;

        if (proc[i].waiting_time < 0)

            proc[i].waiting_time = 0;

    }
```

```c
}

void findTurnaroundTime(struct Process proc[], int n) {

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

        proc[i].turnaround_time = proc[i].burst_time + proc[i].waiting_time;

}

void findAverageTime(struct Process proc[], int n) {

    int total_waiting_time = 0, total_turnaround_time = 0;

    findWaitingTime(proc, n);

    findTurnaroundTime(proc, n);

    printf("Processes  Burst time  Arrival time  Waiting time  Turnaround time\n");

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

        total_waiting_time += proc[i].waiting_time;

        total_turnaround_time += proc[i].turnaround_time;

        printf("   %d \t\t%d \t\t%d \t\t%d \t\t%d\n", proc[i].pid, proc[i].burst_time,
proc[i].arrival_time, proc[i].waiting_time, proc[i].turnaround_time);

    }

    printf("Average waiting time = %.2f\n", (float)total_waiting_time / (float)n);

    printf("Average turnaround time = %.2f\n", (float)total_turnaround_time / (float)n);

}


int main() {

    struct Process proc[] = {{1, 10, 0}, {2, 5, 1}, {3, 8, 2}};

    int n = sizeof(proc) / sizeof(proc[0]);
```

findAverageTime(proc, n);

    return 0;

}

```
Processes   Burst time   Arrival time   Waiting time   Turnaround time
   1            10            0              0                10
   2            5             1              9                14
   3            8             2              13               21
Average waiting time = 7.33
Average turnaround time = 15.00
```

SJF (pre-emptive)

#include <stdio.h>

struct Process {

    int pid;

    int burst_time;

    int arrival_time;

    int waiting_time;

    int turnaround_time;

};

void findWaitingTime(struct Process proc[], int n) {

    int complete = 0, t = 0, minm = 10000;

    int shortest = 0, finish_time;

3

```
int check = 0;

int rt[n];

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

    rt[i] = proc[i].burst_time;


while (complete != n) {

    for (int j = 0; j < n; j++) {

        if ((proc[j].arrival_time <= t) && (rt[j] < minm) && rt[j] > 0) {

            minm = rt[j];

            shortest = j;

            check = 1;

        }

    }

    if (check == 0) {

        t++;

        continue;

    }

    rt[shortest]--;

    minm = rt[shortest];

    if (minm == 0)

        minm = 10000;

    if (rt[shortest] == 0) {

        complete++;
```

```c
        check = 0;

        finish_time = t + 1;

        proc[shortest].waiting_time = finish_time - proc[shortest].burst_time -
proc[shortest].arrival_time;

        if (proc[shortest].waiting_time < 0)

            proc[shortest].waiting_time = 0;

    }

    t++;

  }

}


void findTurnaroundTime(struct Process proc[], int n) {

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

    proc[i].turnaround_time = proc[i].burst_time + proc[i].waiting_time;

}


void findAverageTime(struct Process proc[], int n) {

  int total_waiting_time = 0, total_turnaround_time = 0;


  findWaitingTime(proc, n);

  findTurnaroundTime(proc, n);


  printf("Processes  Burst time  Arrival time  Waiting time  Turnaround time\n");
```

```c
    for (int i = 0; i < n; i++) {

        total_waiting_time += proc[i].waiting_time;

        total_turnaround_time += proc[i].turnaround_time;

        printf("   %d \t\t%d \t\t%d \t\t%d \t\t%d\n", proc[i].pid, proc[i].burst_time,
proc[i].arrival_time, proc[i].waiting_time, proc[i].turnaround_time);

    }


    printf("Average waiting time = %.2f\n", (float)total_waiting_time / (float)n);

    printf("Average turnaround time = %.2f\n", (float)total_turnaround_time / (float)n);

}


int main() {

    struct Process proc[] = {{1, 6, 0}, {2, 8, 1}, {3, 7, 2}, {4, 3, 3}};

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


    findAverageTime(proc, n);


    return 0;

}
```

OUTPUT

```
Processes   Burst time   Arrival time   Waiting time   Turnaround time
   1             6             0             0                6
   2             8             1            15               23
   3             7             2             7               14
   4             3             3             3                6
Average waiting time = 6.25
Average turnaround time = 12.25
```

Program-2

Write a C program to simulate the following CPU scheduling to find turnaround time and waiting time.

Priority (pre-emptive)

```c
#include <stdio.h>

struct Process {

    int pid;

    int burst_time;

    int arrival_time;

    int priority;

    int waiting_time;

    int turnaround_time;

};


void findWaitingTime(struct Process proc[], int n) {

    int rt[n];

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

        rt[i] = proc[i].burst_time;


    int complete = 0, t = 0, minm = 10000;

    int shortest = 0, finish_time;

    int check = 0;


    while (complete != n) {
```

```c
for (int j = 0; j < n; j++) {

    if ((proc[j].arrival_time <= t) && (proc[j].priority < minm) && rt[j] > 0) {

        minm = proc[j].priority;

        shortest = j;

        check = 1;

    }

}

if (check == 0) {

    t++;

    continue;

}

rt[shortest]--;

minm = proc[shortest].priority;

if (rt[shortest] == 0) {

    complete++;

    check = 0;

    finish_time = t + 1;

    proc[shortest].waiting_time = finish_time - proc[shortest].burst_time - proc[shortest].arrival_time;

    if (proc[shortest].waiting_time < 0)

        proc[shortest].waiting_time = 0;

    minm = 10000;

}

t++;
```

```c
    }

}


void findTurnaroundTime(struct Process proc[], int n) {

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

        proc[i].turnaround_time = proc[i].burst_time + proc[i].waiting_time;

}


void findAverageTime(struct Process proc[], int n) {

    int total_waiting_time = 0, total_turnaround_time = 0;


    findWaitingTime(proc, n);

    findTurnaroundTime(proc, n);


    printf("Processes  Burst time  Arrival time  Priority  Waiting time  Turnaround time\n");


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

        total_waiting_time += proc[i].waiting_time;

        total_turnaround_time += proc[i].turnaround_time;

        printf("   %d \t\t%d \t\t%d \t\t%d \t\t%d \t\t%d\n", proc[i].pid, proc[i].burst_time,
proc[i].arrival_time, proc[i].priority, proc[i].waiting_time, proc[i].turnaround_time);

    }


    printf("Average waiting time = %.2f\n", (float)total_waiting_time / (float)n);
```

```c
    printf("Average turnaround time = %.2f\n", (float)total_turnaround_time / (float)n);

}


int main() {

    struct Process proc[] = {{1, 6, 0, 2}, {2, 8, 1, 1}, {3, 7, 2, 3}, {4, 3, 3, 2}};

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


    findAverageTime(proc, n);


    return 0;

}
```

Processes	Burst time	Arrival time	Priority	Waiting time	Turnaround time
1	6	0	2	8	14
2	8	1	1	0	8
3	7	2	3	15	22
4	3	3	2	11	14

Average waiting time = 8.50
Average turnaround time = 14.50

Program 3

Round Robin (Experiment with different quantum sizes for RR algorithm)

#include <stdio.h>

struct Process {

   int pid;

   int burst_time;

   int arrival_time;

   int priority;

   int waiting_time;

   int turnaround_time;

};

void findWaitingTime(struct Process proc[], int n) {

   int completed[n];

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

    completed[i] = 0;

   int t = 0;

   int completed_count = 0;

```
    while (completed_count < n) {

        int min_priority = 10000;

        int idx = -1;

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

            if (proc[i].arrival_time <= t && !completed[i] && proc[i].priority < min_priority) {

                min_priority = proc[i].priority;

                idx = i;

            }

        }


        if (idx != -1) {

            t += proc[idx].burst_time;

            proc[idx].waiting_time = t - proc[idx].burst_time - proc[idx].arrival_time;

            if (proc[idx].waiting_time < 0)

                proc[idx].waiting_time = 0;

            completed[idx] = 1;

            completed_count++;

        } else {

            t++;

        }

    }

}
```

```c
void findTurnaroundTime(struct Process proc[], int n) {

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

        proc[i].turnaround_time = proc[i].burst_time + proc[i].waiting_time;

}


void findAverageTime(struct Process proc[], int n) {

    int total_waiting_time = 0, total_turnaround_time = 0;


    findWaitingTime(proc, n);

    findTurnaroundTime(proc, n);


    printf("Processes  Burst time  Arrival time  Priority  Waiting time  Turnaround time\n");


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

        total_waiting_time += proc[i].waiting_time;

        total_turnaround_time += proc[i].turnaround_time;

        printf("   %d \t\t%d \t\t%d \t\t%d \t\t%d \t\t%d\n", proc[i].pid, proc[i].burst_time,
proc[i].arrival_time, proc[i].priority, proc[i].waiting_time, proc[i].turnaround_time);

    }


    printf("Average waiting time = %.2f\n", (float)total_waiting_time / (float)n);

    printf("Average turnaround time = %.2f\n", (float)total_turnaround_time / (float)n);

}
```

```
int main() {

    struct Process proc[] = {{1, 6, 0, 2}, {2, 8, 1, 1}, {3, 7, 2, 3}, {4, 3, 3, 2}};

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



    findAverageTime(proc, n);



    return 0;

}
```

```
Robin.c -o RoundRobin } ; if ($?) { .\RoundRobin }
Enter the Number of Processes: 3

Enter the quantum time: 2

Enter the process: 1
Enter the Burst Time:4

Enter the process: 2
Enter the Burst Time:3

Enter the process: 3
Enter the Burst Time:5


Processes          Burst Time        Waiting Time       turnaround time

    1                  4                 4                   8

    2                  3                 6                   9

    3                  5                 7                   12

Average waiting time = 5.666667
Average turnaround time = 9.666667
```

Program 4

Write a C program to simulate Real-Time CPU Scheduling algorithms:

Rate- Monotonic

```c
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int period[]) {
    wt[0] = 0;
    for (int i = 1; i < n; i++) {
        wt[i] = bt[i - 1] + wt[i - 1];
    }
}
void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}
void findAvgTime(int processes[], int n, int bt[], int period[]) {
    int wt[n], tat[n];
    findWaitingTime(processes, n, bt, wt, period);
    findTurnAroundTime(processes, n, bt, wt, tat);

    printf("Processes   Burst time   Waiting time   Turnaround time   Period\n");
    for (int i = 0; i < n; i++) {
        printf(" %d ", (i + 1));
        printf("        %d ", bt[i]);
        printf("        %d ", wt[i]);
```

```c
      printf("        %d ", tat[i]);

      printf("              %d\n", period[i]);

   }

   int total_wt = 0, total_tat = 0;

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

      total_wt += wt[i];

      total_tat += tat[i];

   }

   printf("Average waiting time = %.2f\n", (float)total_wt / (float)n);

   printf("Average turnaround time = %.2f\n", (float)total_tat / (float)n);

}

void rateMonotonicScheduling(int processes[], int n, int bt[], int period[]) {

   // Sort by period

   for (int i = 0; i < n - 1; i++) {

      for (int j = 0; j < n - i - 1; j++) {

         if (period[j] > period[j + 1]) {

            int temp = period[j];

            period[j] = period[j + 1];

            period[j + 1] = temp;

            temp = bt[j];

            bt[j] = bt[j + 1];

            bt[j + 1] = temp;

            temp = processes[j];

            processes[j] = processes[j + 1];

            processes[j + 1] = temp;

         }

      }
```

```
    }


  findAvgTime(processes, n, bt, period);

}


int main() {

    int processes[] = {1, 2, 3};

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

    int burst_time[] = {3, 1, 2};

    int period[] = {7, 4, 5};


    rateMonotonicScheduling(processes, n, burst_time, period);

    return 0;

}
```

```
Processes    Burst time    Waiting time    Turnaround time    Period
1            1             0               1                  4
2            2             1               3                  5
3            3             3               6                  7
Average waiting time = 1.33
Average turnaround time = 3.33
```

Earliest-deadline First

```
#include <stdio.h>


void findWaitingTime(int processes[], int n, int bt[], int wt[], int deadline[]) {

    wt[0] = 0;

    for (int i = 1; i < n; i++) {

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

```c
    }

}


void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

      tat[i] = bt[i] + wt[i];

   }

}


void findAvgTime(int processes[], int n, int bt[], int deadline[]) {

   int wt[n], tat[n];

   findWaitingTime(processes, n, bt, wt, deadline);

   findTurnAroundTime(processes, n, bt, wt, tat);


   printf("Processes   Burst time   Waiting time   Turnaround time   Deadline\n");

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

      printf(" %d ", (i + 1));

      printf("        %d ", bt[i]);

      printf("        %d ", wt[i]);

      printf("        %d ", tat[i]);

      printf("          %d\n", deadline[i]);

   }


   int total_wt = 0, total_tat = 0;

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

      total_wt += wt[i];

      total_tat += tat[i];
```

```c
    }

    printf("Average waiting time = %.2f\n", (float)total_wt / (float)n);

    printf("Average turnaround time = %.2f\n", (float)total_tat / (float)n);

}


void earliestDeadlineFirstScheduling(int processes[], int n, int bt[], int deadline[]) {
    // Sort by deadline
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (deadline[j] > deadline[j + 1]) {
                int temp = deadline[j];
                deadline[j] = deadline[j + 1];
                deadline[j + 1] = temp;
                temp = bt[j];
                bt[j] = bt[j + 1];
                bt[j + 1] = temp;
                temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }


    findAvgTime(processes, n, bt, deadline);
}
```

```
int main() {

    int processes[] = {1, 2, 3};

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

    int burst_time[] = {3, 1, 2};

    int deadline[] = {7, 4, 5};


    earliestDeadlineFirstScheduling(processes, n, burst_time, deadline);

    return 0;

}
```

```
Processes    Burst time    Waiting time    Turnaround time    Ratio
1            3             0               3                  0.50
2            2             3               5                  0.30
3            1             5               6                  0.20
Average waiting time = 2.67
Average turnaround time = 4.67
```

Program 5

Write a C program to simulate producer-consumer problem using semaphores.

```c
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>


#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];

int in = 0, out = 0;


sem_t empty;

sem_t full;

pthread_mutex_t mutex;


void *producer(void *param) {

   int item;

   while (1) {

      item = rand() % 100;

      sem_wait(&empty);

      pthread_mutex_lock(&mutex);


      buffer[in] = item;

      printf("Producer produced %d at %d\n", item, in);

      in = (in + 1) % BUFFER_SIZE;
```

```
        pthread_mutex_unlock(&mutex);

        sem_post(&full);

        sleep(1);

    }

}


void *consumer(void *param) {

    int item;

    while (1) {

        sem_wait(&full);

        pthread_mutex_lock(&mutex);


        item = buffer[out];

        printf("Consumer consumed %d from %d\n", item, out);

        out = (out + 1) % BUFFER_SIZE;


        pthread_mutex_unlock(&mutex);

        sem_post(&empty);

        sleep(1);

    }

}


int main() {

    pthread_t tid1, tid2;

    pthread_attr_t attr;
```

```c
    pthread_attr_init(&attr);

    pthread_mutex_init(&mutex, NULL);

    sem_init(&empty, 0, BUFFER_SIZE);

    sem_init(&full, 0, 0);


    pthread_create(&tid1, &attr, producer, NULL);

    pthread_create(&tid2, &attr, consumer, NULL);


    pthread_join(tid1, NULL);

    pthread_join(tid2, NULL);


    pthread_mutex_destroy(&mutex);

    sem_destroy(&empty);

    sem_destroy(&full);


    return 0;

}
```

```
                |              ^~~~~
Producer produced 83 at 0
Consumer consumed 83 from 0
Producer produced 86 at 1
Consumer consumed 86 from 1
Producer produced 77 at 2
Consumer consumed 77 from 2
Producer produced 15 at 3
Consumer consumed 15 from 3
Producer produced 93 at 4
Consumer consumed 93 from 4
Producer produced 35 at 0
Consumer consumed 35 from 0
Producer produced 86 at 1
Consumer consumed 86 from 1
Producer produced 92 at 2
Consumer consumed 92 from 2
Producer produced 49 at 3
Consumer consumed 49 from 3
Producer produced 21 at 4
Consumer consumed 21 from 4
Producer produced 62 at 0
Consumer consumed 62 from 0
Producer produced 27 at 1
Consumer consumed 27 from 1
Producer produced 90 at 2
Consumer consumed 90 from 2
```

Program 6

Write a C program to simulate the concept of Dining-Philosophers problem.

```c
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>


#define N

sem_t forks[N];

sem_t mutex;


void *philosopher(void *num) {

  int id = *(int *)num;

  while (1) {

    printf("Philosopher %d is thinking.\n", id);

    sleep(1);


    sem_wait(&mutex);

    sem_wait(&forks[id]);

    sem_wait(&forks[(id + 1) % N]);


    printf("Philosopher %d is eating.\n", id);

    sleep(1);


    sem_post(&forks[id]); // Put down chopsticks
```

```c
        sem_post(&forks[(id + 1) % N]);

        sem_post(&mutex);


        printf("Philosopher %d is done eating and starts thinking again.\n", id);

        sleep(1);

    }

}


int main() {

    pthread_t tid[N];

    int ids[N];


    sem_init(&mutex, 0, 1);


    for (int i = 0; i < N; i++) {

        sem_init(&forks[i], 0, 1);

        ids[i] = i;

    }


    for (int i = 0; i < N; i++) {

        pthread_create(&tid[i], NULL, philosopher, &ids[i]);

    }


    for (int i = 0; i < N; i++) {

        pthread_join(tid[i], NULL);

    }
```

```
    for (int i = 0; i < N; i++) {

        sem_destroy(&forks[i]);

    }

    sem_destroy(&mutex);


    return 0;

}
```

```
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 0 is eating.
Philosopher 0 is done eating and starts thinking again.
Philosopher 1 is eating.
Philosopher 0 is thinking.
Philosopher 1 is done eating and starts thinking again.
Philosopher 2 is eating.
Philosopher 3 is eating.
Philosopher 2 is done eating and starts thinking again.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 4 is eating.
Philosopher 3 is done eating and starts thinking again.
Philosopher 3 is thinking.
Philosopher 0 is eating.
Philosopher 4 is done eating and starts thinking again.
Philosopher 4 is thinking.
Philosopher 1 is eating.
Philosopher 0 is done eating and starts thinking again.
Philosopher 1 is done eating and starts thinking again.
Philosopher 2 is eating.
Philosopher 0 is thinking.
Philosopher 2 is done eating and starts thinking again.
Philosopher 1 is thinking.
Philosopher 3 is eating.
Philosopher 2 is thinking.
Philosopher 4 is eating.
Philosopher 3 is done eating and starts thinking again.
```

Program 7

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

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

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

int main() {
  int n, m, i, j, k;
  n = 5;
  m = 3;

  int alloc[MAX_PROCESSES][MAX_RESOURCES] = { { 0, 1, 0 },
                    { 2, 0, 0 },
                    { 3, 0, 2 },
                    { 2, 1, 1 },
                    { 0, 0, 2 } };

  int max[MAX_PROCESSES][MAX_RESOURCES] = { { 7, 5, 3 },
                    { 3, 2, 2 },
                    { 9, 0, 2 },
                    { 2, 2, 2 },
                    { 4, 3, 3 } };
```

```c
int avail[MAX_RESOURCES] = { 3, 3, 2 };

int f[MAX_PROCESSES], ans[MAX_PROCESSES], ind = 0;

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

    f[k] = 0;

}


int need[MAX_PROCESSES][MAX_RESOURCES];

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

    for (j = 0; j < m; j++) {

        need[i][j] = max[i][j] - alloc[i][j];

    }

}

printf("Need matrix:\n");

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

    for (j = 0; j < m; j++) {

        printf("%d ", need[i][j]);

    }

    printf("\n");

}

int y = 0;

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

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

        if (f[i] == 0) {

            bool flag = true;

            for (j = 0; j < m; j++) {

                if (need[i][j] > avail[j]) {

                    flag = false;
```

```
                break;
            }   }
        if (flag) {
            ans[ind++] = i;
            for (y = 0; y < m; y++) {
                avail[y] += alloc[i][y];
            }
            f[i] = 1;
        }
    }
}
printf("Following is the SAFE Sequence:\n");
for (i = 0; i < n - 1; i++) {
    printf(" P%d ->", ans[i]);
}
printf(" P%d\n", ans[n - 1]);


return 0;
}
```

```
Need matrix:
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1
Following is the SAFE Sequence:
 P1 -> P3 -> P4 -> P0 -> P2
```

Program 8

Write a C program to simulate deadlock detection

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

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

void printMatrices(int processes, int resources, int alloc[MAX_PROCESSES][MAX_RESOURCES],
int max[MAX_PROCESSES][MAX_RESOURCES], int need[MAX_PROCESSES][MAX_RESOURCES],
int avail[MAX_RESOURCES]) {
    printf("Allocation Matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            printf("%d ", alloc[i][j]);
        }
        printf("\n");
    }

    printf("\nMax Matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            printf("%d ", max[i][j]);
        }
        printf("\n");
    }
```

```c
    printf("\nNeed Matrix:\n");

    for (int i = 0; i < processes; i++) {

        for (int j = 0; j < resources; j++) {

            printf("%d ", need[i][j]);

        }

        printf("\n");

    }


    printf("\nAvailable Resources:\n");

    for (int i = 0; i < resources; i++) {

        printf("%d ", avail[i]);

    }

    printf("\n");

}


void deadlockDetection(int processes, int resources, int
alloc[MAX_PROCESSES][MAX_RESOURCES], int max[MAX_PROCESSES][MAX_RESOURCES], int
avail[MAX_RESOURCES]) {

    int need[MAX_PROCESSES][MAX_RESOURCES];

    int work[MAX_RESOURCES];

    bool finish[MAX_PROCESSES];


    for (int i = 0; i < processes; i++) {

        for (int j = 0; j < resources; j++) {

            need[i][j] = max[i][j] - alloc[i][j];

        }

    }
```

```c
printMatrices(processes, resources, alloc, max, need, avail);

for (int i = 0; i < resources; i++) {
    work[i] = avail[i];
}

for (int i = 0; i < processes; i++) {
    finish[i] = false;
}

bool found;

do {
    found = false;
    for (int i = 0; i < processes; i++) {
        if (!finish[i]) {
            bool flag = true;
            for (int j = 0; j < resources; j++) {
                if (need[i][j] > work[j]) {
                    flag = false;
                    break;
                }
            }

            if (flag) {
                printf("\nProcess %d can be satisfied and is now finishing.\n", i);
                for (int k = 0; k < resources; k++) {
                    work[k] += alloc[i][k];
                }
                finish[i] = true;
                found = true;
```

```c
            printf("New Available Resources:\n");

            for (int k = 0; k < resources; k++) {

                printf("%d ", work[k]);

            }

            printf("\n");

        }

      }

    }

  } while (found);

  bool deadlock = false;

  printf("\nDeadlock Check:\n");

  for (int i = 0; i < processes; i++) {

    if (!finish[i]) {

      deadlock = true;

      printf("Process %d is in a deadlock.\n", i);

    }

  }

  if (!deadlock) {

    printf("No deadlock detected.\n");

  }

}

int main() {

  int processes = 5;

  int resources = 3;


  int alloc[MAX_PROCESSES][MAX_RESOURCES] = {
```

```
        { 0, 1, 0 },

        { 2, 0, 0 },

        { 3, 0, 2 },

        { 2, 1, 1 },

        { 0, 0, 2 }

    };

    int max[MAX_PROCESSES][MAX_RESOURCES] = {

        { 7, 5, 3 },

        { 3, 2, 2 },

        { 9, 0, 2 },

        { 2, 2, 2 },

        { 4, 3, 3 }

    };

    int avail[MAX_RESOURCES] = { 3, 3, 2 }; // Available resources

    deadlockDetection(processes, resources, alloc, max, avail);

    return 0;

}
```

```
Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2

Max Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

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

Available Resources:
3 3 2
```

```
Process 1 can be satisfied and is now finishing.
New Available Resources:
5 3 2

Process 3 can be satisfied and is now finishing.
New Available Resources:
7 4 3

Process 4 can be satisfied and is now finishing.
New Available Resources:
7 4 5

Process 0 can be satisfied and is now finishing.
New Available Resources:
7 5 5

Process 2 can be satisfied and is now finishing.
New Available Resources:
10 5 7

Deadlock Check:
No deadlock detected.
```

Program 9

Write a C program to simulate the following contiguous memory allocation techniques

a) Worst-fit

b) Best-fit

c) First-fit

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

#define MAX 25

void firstFit(int nb, int nf, int b[], int f[]) {
    int allocation[MAX];
    int allocated[MAX] = {0};
    for (int i = 0; i < nf; i++) {
        allocation[i] = -1;
        for (int j = 0; j < nb; j++) {
            if (allocated[j] == 0 && b[j] >= f[i]) {
                allocation[i] = j;
                allocated[j] = 1;
                break;
            }
        }
    }

    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:");
```

```c
    for (int i = 0; i < nf; i++) {

        if (allocation[i] != -1)

            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], allocation[i] + 1, b[allocation[i]]);

        else

            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);

    }

}


void bestFit(int nb, int nf, int b[], int f[]) {

    int allocation[MAX];

    int allocated[MAX] = {0};

    for (int i = 0; i < nf; i++) {

        int bestIdx = -1;

        allocation[i] = -1;

        for (int j = 0; j < nb; j++) {

            if (allocated[j] == 0 && b[j] >= f[i]) {

                if (bestIdx == -1 || b[j] < b[bestIdx])

                    bestIdx = j;

            }

        }

        if (bestIdx != -1) {

            allocation[i] = bestIdx;

            allocated[bestIdx] = 1;

        }

    }


    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:");
```

```c
    for (int i = 0; i < nf; i++) {

        if (allocation[i] != -1)

            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], allocation[i] + 1, b[allocation[i]]);

        else

            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);

    }

}


void worstFit(int nb, int nf, int b[], int f[]) {

    int allocation[MAX];

    int allocated[MAX] = {0};


    for (int i = 0; i < nf; i++) {

        int worstIdx = -1;

        allocation[i] = -1;

        for (int j = 0; j < nb; j++) {

            if (allocated[j] == 0 && b[j] >= f[i]) {

                if (worstIdx == -1 || b[j] > b[worstIdx])

                    worstIdx = j;

            }

        }

        if (worstIdx != -1) {

            allocation[i] = worstIdx;

            allocated[worstIdx] = 1;

        }

    }
```

```c
    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:");

    for (int i = 0; i < nf; i++) {

        if (allocation[i] != -1)

            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], allocation[i] + 1, b[allocation[i]]);

        else

            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);

    }

}


int main() {

    int nb, nf, choice;


    printf("Memory Management Scheme");

    printf("\nEnter the number of blocks: ");

    scanf("%d", &nb);

    printf("Enter the number of files: ");

    scanf("%d", &nf);

    int b[nb], f[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("Enter the size of the files:\n");

    for (int i = 0; i < nf; i++) {

        printf("File %d: ", i + 1);

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

41

```c
    }

    while (1) {
        printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\n\tMemory Management Scheme - First Fit\n");
                firstFit(nb, nf, b, f);
                break;
            case 2:
                printf("\n\tMemory Management Scheme - Best Fit\n");
                bestFit(nb, nf, b, f);
                break;
            case 3:
                printf("\n\tMemory Management Scheme - Worst Fit\n");
                worstFit(nb, nf, b, f);
                break;
            case 4:
                printf("\nExiting...\n");
                exit(0);
                break;
            default:
                printf("\nInvalid choice.\n");
                break;
        }
```

```
    }


    return 0;

}
```

```
Memory Management Scheme
Enter the number of blocks: 5
Enter the number of files: 4

Enter the size of the blocks:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600
Enter the size of the files:
File 1: 212
File 2: 417
File 3: 112
File 4: 426
```

```
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 1

        Memory Management Scheme - First Fit

File_no:          File_size:        Block_no:        Block_size:
1                 212               2                500
2                 417               5                600
3                 112               3                200
4                 426               -                -
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 2

        Memory Management Scheme - 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
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 3

        Memory Management Scheme - Worst Fit

File_no:          File_size:        Block_no:        Block_size:
1                 212               5                600
2                 417               2                500
3                 112               4                300
4                 426               -                -
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice:
```

Program 10

Write a C program to simulate page replacement algorithms

a) FIFO

b) LRU

c) Optimal

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

#define MAX_FRAMES 10
#define MAX_PAGES 25

void fifo(int pages[], int n, int capacity) {
    int frame[MAX_FRAMES], frameCount = 0, pageFaults = 0, frameIndex = 0;
    bool isPagePresent = false;

    for (int i = 0; i < n; i++) {
        isPagePresent = false;
        for (int j = 0; j < frameCount; j++) {
            if (frame[j] == pages[i]) {
                isPagePresent = true;
                break;
            }
        }
    }
```

```c
        if (isPagePresent == false) {

            if (frameCount < capacity) {

                frame[frameCount] = pages[i];

                frameCount++;

            } else {

                frame[frameIndex] = pages[i];

                frameIndex++;

                if (frameIndex >= capacity)

                    frameIndex = 0;

            }

            pageFaults++;

        }

    }

    printf("\nFIFO Page Replacement Algorithm:\n");

    printf("Total Page Faults: %d\n", pageFaults);

}


void lru(int pages[], int n, int capacity) {

    int frame[MAX_FRAMES], frameCount = 0, pageFaults = 0, counter[MAX_FRAMES];

    bool isPagePresent = false;


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

        isPagePresent = false;

        for (int j = 0; j < frameCount; j++) {

            if (frame[j] == pages[i]) {

                isPagePresent = true;

                counter[j] = i;
```

```c
            break;

        }

    }


    if (isPagePresent == false) {

        if (frameCount < capacity) {

            frame[frameCount] = pages[i];

            counter[frameCount] = i;

            frameCount++;

        } else {

            int lru = 0;

            for (int j = 1; j < capacity; j++) {

                if (counter[j] < counter[lru])

                    lru = j;

            }

            frame[lru] = pages[i];

            counter[lru] = i;

        }

        pageFaults++;

    }

}

printf("\nLRU Page Replacement Algorithm:\n");

printf("Total Page Faults: %d\n", pageFaults);

}


void optimal(int pages[], int n, int capacity) {

    int frame[MAX_FRAMES], frameCount = 0, pageFaults = 0;
```

```
bool isPagePresent = false;


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

    isPagePresent = false;

    for (int j = 0; j < frameCount; j++) {

        if (frame[j] == pages[i]) {

            isPagePresent = true;

            break;

        }

    }


    if (isPagePresent == false) {

        if (frameCount < capacity) {

            frame[frameCount] = pages[i];

            frameCount++;

        } else {

            int future[MAX_FRAMES] = {0};

            for (int j = 0; j < frameCount; j++) {

                bool isFound = false;

                for (int k = i + 1; k < n; k++) {

                    if (pages[k] == frame[j]) {

                        future[j] = k;

                        isFound = true;

                        break;

                    }

                }

                if (isFound == false)
```

```c
            future[j] = n + 1;
        }

        int longest = 0;
        for (int j = 1; j < frameCount; j++) {
            if (future[j] > future[longest])
                longest = j;
        }

        frame[longest] = pages[i];
    }

    pageFaults++;
  }
}
printf("\nOptimal Page Replacement Algorithm:\n");
printf("Total Page Faults: %d\n", pageFaults);
}


int main() {
  int pages[MAX_PAGES], n, capacity;

  printf("Page Replacement Algorithms\n");
  printf("Enter the number of pages: ");
  scanf("%d", &n);
  printf("Enter the page reference string:\n");
  for (int i = 0; i < n; i++) {
    printf("Page %d: ", i + 1);
    scanf("%d", &pages[i]);
  }
```

```c
    printf("Enter the number of frames: ");

    scanf("%d", &capacity);


    fifo(pages, n, capacity);

    lru(pages, n, capacity);

    optimal(pages, n, capacity);


    return 0;

}
```

```
Page Replacement Algorithms
Enter the number of pages: 10
Enter the page reference string:
Page 1: 1
Page 2: 2
Page 3: 1
Page 4: 4
Page 5: 6
Page 6: 4
Page 7: 2
Page 8: 1
Page 9: 56
Page 10: 3
Enter the number of frames: 3

FIFO Page Replacement Algorithm:
Total Page Faults: 7

LRU Page Replacement Algorithm:
Total Page Faults: 8

Optimal Page Replacement Algorithm:
Total Page Faults: 7
```

Write a C program to simulate disk scheduling algorithms:

(a) FCFS

(b) SCAN

(c) c-SCAN

(a) FCFS:

```c
#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

printf("Enter initial head position\n");

scanf("%d",&initial);

// logic for FCFS disk scheduling

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

printf("Total head moment is %d",TotalHeadMoment);

return 0;

}
```

```
Enter the number of Requests
8
Enter the Requests sequence
98 183 37 122 14 124 65 67
Enter initial head position
53
Total head moment is 640
```

(b) SCAN:

```
#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");


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

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

printf("Enter initial head position\n");

scanf("%d",&initial);

printf("Enter total disk size\n");

scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n");

scanf("%d",&move);

// logic for Scan disk scheduling

/*logic for sort the request array */

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

```
{
for(j=0;j<n-i-1;j++)
{
if(RQ[j]>RQ[j+1])
{
int temp;
temp=RQ[j];
RQ[j]=RQ[j+1];
RQ[j+1]=temp;
}
}
}
int index;
for(i=0;i<n;i++)
{
if(initial<RQ[i])
{
index=i;
break;
}
}
//if movement is towards high value
if(move==1)
{
for(i=index;i<n;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
```

```
initial=RQ[i];

}

// last movement for max size

TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);

initial = size-1;

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

// if movement is towards low value

else

{

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for min size

TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

initial =0;

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];
```

```
}

}
```

printf("Total head movement is %d",TotalHeadMoment);

return 0;

```
}
```

```
Enter the number of Requests
8
Enter the Requests sequence
98 183 37 122 14 124 65 67
Enter initial head position
53
Enter total disk size
199
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 236
```

(c) c-SCAN:

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

printf("Enter initial head position\n");

```c
scanf("%d",&initial);

printf("Enter total disk size\n");

scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n");

scanf("%d",&move);

// logic for C-Scan disk scheduling

/*logic for sort the request array */

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

{

for( j=0;j<n-i-1;j++)

{

if(RQ[j]>RQ[j+1])

{

int temp;

temp=RQ[j];

RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

int index;

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

{

if(initial<RQ[i])

{

index=i;

break;
```

```
}

}

// if movement is towards high value

if(move==1)

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);


initial=RQ[i];

}

// last movement for max size

TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);

/*movement max to min disk */

TotalHeadMoment=TotalHeadMoment+abs(size-1-0);

initial=0;

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

// if movement is towards low value

else

{

for(i=index-1;i>=0;i--)

{
```

```c
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for min size

TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

/*movement min to max disk */

TotalHeadMoment=TotalHeadMoment+abs(size-1-0);

initial =size-1;

for(i=n-1;i>=index;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

printf("Total head movement is %d",TotalHeadMoment);

return 0;

}
```

```
Enter the number of Requests
8
Enter the Requests sequence
98 183 37 122 14 124 65 67
Enter initial head position
53
Enter total disk size
199
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 384
```