**Report On**

# Operating System (CSC-264)

**<u>Submitted To</u>**

**Mrs. Amrita Thapa**

**<u>Submitted By</u>**

**Sulav Baral**

**Roll no : 09 (Sec.A)**

**Symbol no: 80012181**

INDEX

| SN | Experiment | Date of Experiment | Date of Submission | Remarks |
|---|---|---|---|---|
| 1. | To demonstrate process creation using the fork() system call using C. | 2082-03-27 | 2082-04-05 | |
| 2. | To show the implementation of Round Robin Algorithm using C. | 2082-03-27 | 2082-04-05 | |
| 3. | To show the implementation of First Come First Serve Algorithm using C. | 2082-03-27 | 2082-04-05 | |
| 4. | To show the implementation of Shortest Job First Algorithm using C. | 2082-03-27 | 2082-04-05 | |
| | | | | |

## Lab:1

**OBJECTIVE:** To demonstrate process creation using the fork() system call using C.

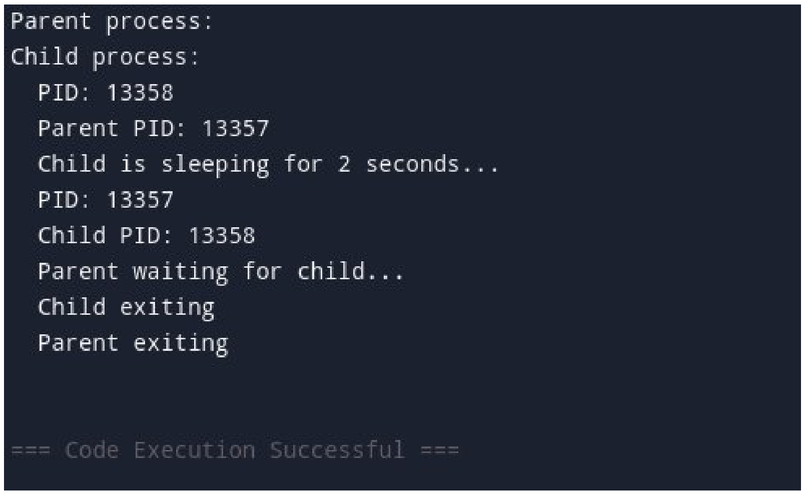
### THEORY:

The fork() system call in C is a fundamental function used in UNIX-like operating systems to create new processes. When a process calls fork(), it creates a duplicate of itself known as the child process, while the original becomes the parent process. Both processes continue execution independently from the point where fork() was called. The return value of fork() helps distinguish between the two: the child receives a return value of 0, and the parent receives the PID of the child. If the call fails, it returns -1 to the parent. The child process inherits the memory space, environment, and open file descriptors of the parent, but operates in a separate address space. This mechanism is crucial for multitasking, running background processes, or initiating new program executions using exec() functions. Proper handling is required to manage process termination and avoid zombie or orphan processes, typically using functions like wait() or waitpid(). Thus, fork() plays a vital role in process management and is a key concept in systems programming.

### SOURCE CODE:

```
#include <stdio.h>
#include <unistd.h>  // Required for fork() and sleep()
#include <sys/types.h> // Required for pid_t
#include <sys/wait.h>  // Required for wait()

int main() {
    pid_t pid;  // Process ID variable

    // Create a new process
    pid = fork();

    // Check for fork error
    if (pid < 0) {
        fprintf(stderr, "Fork failed\n");
        return 1;
    }
    // Child process
    if (pid == 0) {
        printf("Child process:\n");
        printf("  PID: %d\n", getpid());
        printf("  Parent PID: %d\n", getppid());
        printf("  Child is sleeping for 2 seconds...\n");
        sleep(2);  // Simulate some work
        printf("  Child exiting\n");
    }
    // Parent process
    else {
        printf("Parent process:\n");
        printf("  PID: %d\n", getpid());
        printf("  Child PID: %d\n", pid);
        printf("  Parent waiting for child...\n");
        wait(NULL);  // Wait for child to finish
        printf("  Parent exiting\n");
    }
    return 0;
}
```

## OUTPUT:



```
Parent process:
Child process:
  PID: 13358
  Parent PID: 13357
  Child is sleeping for 2 seconds...
  PID: 13357
  Child PID: 13358
  Parent waiting for child...
  Child exiting
  Parent exiting


=== Code Execution Successful ===
```

## CONCLUSION:

The program successfully demonstrates the creation of a child process using the fork() system call. It shows how both parent and child processes can run independently and how the parent can synchronize with the child using the wait() function. This ensures the parent waits until the child completes execution. The use of getpid() and getppid() helps in identifying process relationships,while sleep() simulates a delay to mimic real processing. This program provides a clear understanding of process creation, execution order, and synchronization in a Unix-based operating system.

## Lab:2

**OBJECTIVE:** To show the implementation of Round Robin Algorithm using C.

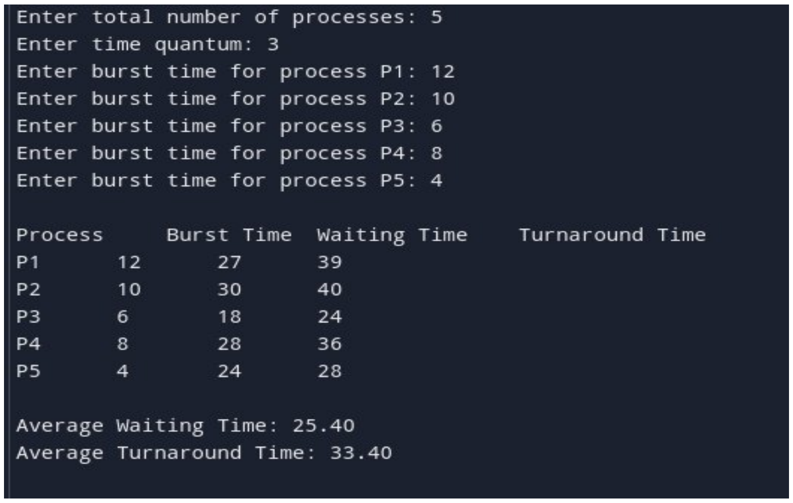
### THEORY:

The Round Robin (RR) Scheduling Algorithm is a preemptive CPU scheduling technique primarily used in time-sharing systems. In this algorithm, each process is assigned a fixed time slot called a time quantum. The processes are executed in a circular queue order. If a process does not finish during its allotted quantum, it is paused and placed at the back of the queue, allowing the next process to run. This cycle continues until all processes are completed. Round Robin ensures fairness by giving each process equal CPU access and minimizes starvation. It is ideal for systems where response time is more critical than throughput. The program uses arrays to store burst time, waiting time, and turnaround time for each process and simulates RR scheduling to calculate the average waiting and turnaround times.

### SOURCE CODE:

```
#include <stdio.h>
int main() {
    int n, quantum;
    printf("Enter total number of processes: ");
    scanf("%d", &n);
    printf("Enter time quantum: ");
    scanf("%d", &quantum);
 int burst_time[n], waiting_time[n], turnaround_time[n], remaining_time[n];
    int i, time = 0, completed = 0;
  // Input burst times
    for(i = 0; i < n; i++) {
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &burst_time[i]);
        remaining_time[i] = burst_time[i];
        waiting_time[i] = 0;
    }
// Round Robin scheduling
    while(completed < n) {
        for(i = 0; i < n; i++) {
            if(remaining_time[i] > 0) {
                if(remaining_time[i] > quantum) {
                    // Process executes for full quantum
                    time += quantum;
                    remaining_time[i] -= quantum;
                } else {
                    // Process completes execution
                    time += remaining_time[i];
                    waiting_time[i] = time - burst_time[i];
                    remaining_time[i] = 0;
                    completed++;
                }    }   } }
  // Calculate turnaround times
    float avg_waiting = 0, avg_turnaround = 0;
    for(i = 0; i < n; i++) {
        turnaround_time[i] = burst_time[i] + waiting_time[i];
        avg_waiting += waiting_time[i];
        avg_turnaround += turnaround_time[i];
    }   // Calculate averages
    avg_waiting /= n;
    avg_turnaround /= n;

    // Display results
    printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for(i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\n", i+1, burst_time[i], waiting_time[i], turnaround_time[i]);
    }
 printf("\nAverage Waiting Time: %.2f\n", avg_waiting);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround);
  return 0;}
```

## OUTPUT:



```
Enter total number of processes: 5
Enter time quantum: 3
Enter burst time for process P1: 12
Enter burst time for process P2: 10
Enter burst time for process P3: 6
Enter burst time for process P4: 8
Enter burst time for process P5: 4

Process     Burst Time  Waiting Time    Turnaround Time
P1      12      27      39
P2      10      30      40
P3      6       18      24
P4      8       28      36
P5      4       24      28

Average Waiting Time: 25.40
Average Turnaround Time: 33.40
```

## CONCLUSION:

The implementation of the Round Robin algorithm successfully demonstrates how each process gets equal CPU time in a rotating manner. The program calculates the waiting and turnaround times for each process based on the time quantum and burst times. This experiment highlights the fairness and efficiency of Round Robin in managing multiple processes in a time-shared environment. It is particularly suitable for applications requiring predictable response times and demonstrates how preemptive scheduling works in operating systems.

## Lab:3

**OBJECTIVE:** To show the implementation of First Come First Serve Algorithm**.**

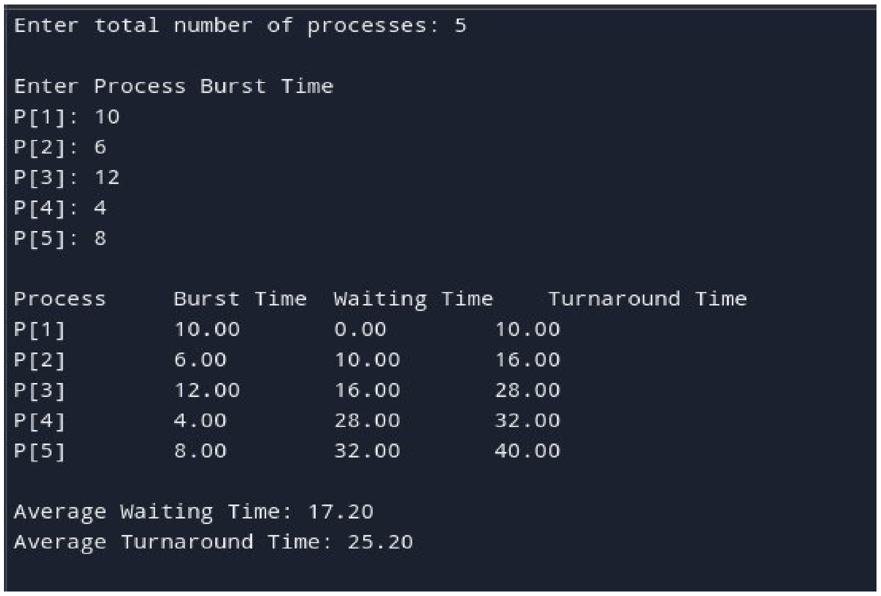
## THEORY:

First-Come, First-Served (FCFS) is one of the simplest types of CPU scheduling algorithms. In this method, the process that arrives first is executed first without preemption. The CPU is allocated in the order of arrival, making it a non-preemptive scheduling technique. Each process must wait until all earlier-arriving processes finish their execution. The waiting time for each process is the sum of the burst times of all the preceding processes. This algorithm is easy to implement using a queue data structure. However, FCFS can lead to problems like convoy effect, where short processes wait for a long process to finish, leading to increased average waiting time. In the given C program, the burst times are collected, and then the waiting time and turnaround time for each process are calculated. The results are displayed along with the average waiting and turnaround times.

## SOURCE CODE:

```
#include<stdio.h>
int main() {
    int n, i, j;
    float bt[20], wt[20], tat[20];
    float avwt = 0, avtat = 0;
    printf("Enter total number of processes: ");
    scanf("%d", &n);
    printf("\nEnter Process Burst Time\n");
    for(i = 0; i < n; i++) {
        printf("P[%d]: ", i+1);
        scanf("%f", &bt[i]);
    }
    wt[0] = 0;
    for(i = 1; i < n; i++) {
        wt[i] = 0;
        for(j = 0; j < i; j++)
            wt[i] += bt[j];
    }
     printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
    for(i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
        avwt += wt[i];
        avtat += tat[i];
        printf("\nP[%d]\t\t%.2f\t\t%.2f\t\t%.2f", i+1, bt[i], wt[i], tat[i]);
    }
    avwt /= n;
    avtat /= n;
    printf("\n\nAverage Waiting Time: %.2f", avwt);
    printf("\nAverage Turnaround Time: %.2f", avtat);
    return 0;
}
```

## OUTPUT:



```
Enter total number of processes: 5

Enter Process Burst Time
P[1]: 10
P[2]: 6
P[3]: 12
P[4]: 4
P[5]: 8

Process     Burst Time  Waiting Time    Turnaround Time
P[1]        10.00       0.00        10.00
P[2]        6.00        10.00       16.00
P[3]        12.00       16.00       28.00
P[4]        4.00        28.00       32.00
P[5]        8.00        32.00       40.00

Average Waiting Time: 17.20
Average Turnaround Time: 25.20
```

## CONCLUSION:

The FCFS scheduling algorithm was successfully implemented in C, showing how processes are handled in the order they arrive. The program calculated and displayed the waiting time and turnaround time for each process based on their burst times. The results demonstrated the simplicity of FCFS but also highlighted its limitations, such as high average waiting time in certain cases. Overall, this experiment provided a clear understanding of how non-preemptive scheduling works and how process order impacts overall performance.

## Lab:4

**OBJECTIVE:** To show the implementation of Shortest Job First Algorithm.

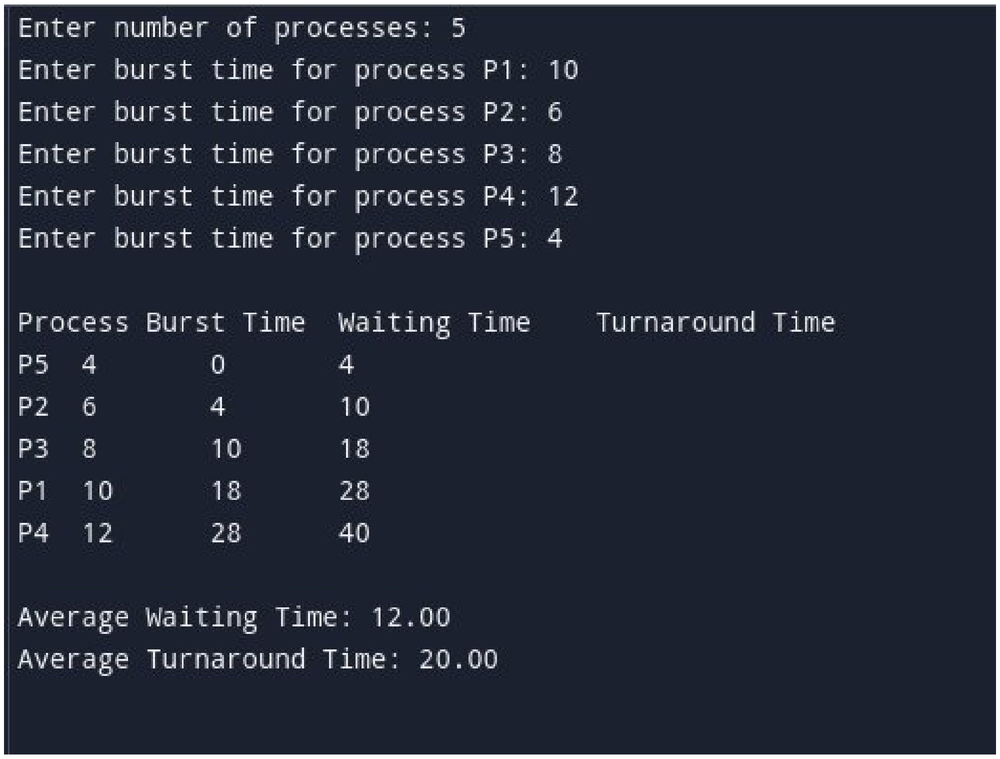
### THEORY:

Shortest Job First (SJF) is a non-preemptive scheduling algorithm that selects the process with the shortest burst time to execute next. It minimizes average waiting time, making it one of the most optimal scheduling techniques under ideal conditions. In SJF, all processes are sorted in ascending order of their burst times before scheduling begins. The process with the shortest burst time is executed first, followed by the next shortest, and so on. Waiting time is calculated by summing the burst times of all previous processes, and turnaround time is the sum of waiting time and burst time for each process. The algorithm assumes that all processes arrive at the same time and that burst times are known in advance. The provided C program implements this logic using a structure to store process information and a simple sorting algorithm to arrange processes based on burst time. It then calculates and displays the waiting time, turnaround time, and their averages.

### SOURCE CODE:

```
#include <stdio.h>
// Structure to represent a process
struct Process {
    int pid;        // Process ID
    int burst;      // Burst time
    int waiting;    // Waiting time
    int turnaround; // Turnaround time
};
int main() {
    int n, i, j;
    float avg_waiting = 0, avg_turnaround = 0;
     printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process proc[n], temp;
     // Input burst times for each process
    for(i = 0; i < n; i++) {
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &proc[i].burst);
        proc[i].pid = i + 1;  // Assign process IDs
    }
      // Sort processes by burst time (SJF)
    for(i = 0; i < n - 1; i++) {
        for(j = 0; j < n - i - 1; j++) {
            if(proc[j].burst > proc[j + 1].burst) {
                // Swap processes
                temp = proc[j];
                proc[j] = proc[j + 1];
                proc[j + 1] = temp;
            }  }  }
     // Calculate waiting and turnaround times
    proc[0].waiting = 0;  // First process has 0 waiting time
    proc[0].turnaround = proc[0].burst;
    for(i = 1; i < n; i++) {
        proc[i].waiting = proc[i - 1].turnaround;
        proc[i].turnaround = proc[i].waiting + proc[i].burst;
    }
    // Calculate averages
    for(i = 0; i < n; i++) {
        avg_waiting += proc[i].waiting;
        avg_turnaround += proc[i].turnaround;
    }
     avg_waiting /= n;
    avg_turnaround /= n;
    // Print results
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for(i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\n",
               proc[i].pid, proc[i].burst, proc[i].waiting, proc[i].turnaround);
    }
    printf("\nAverage Waiting Time: %.2f", avg_waiting);
    printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround);
    return 0;
}
```

## OUTPUT:



```
Enter number of processes: 5
Enter burst time for process P1: 10
Enter burst time for process P2: 6
Enter burst time for process P3: 8
Enter burst time for process P4: 12
Enter burst time for process P5: 4

Process Burst Time  Waiting Time    Turnaround Time
P5  4       0       4
P2  6       4       10
P3  8       10      18
P1  10      18      28
P4  12      28      40

Average Waiting Time: 12.00
Average Turnaround Time: 20.00
```

## CONCLUSION:

The Shortest Job First (SJF) scheduling algorithm was effectively implemented using C, demonstrating how CPU scheduling based on burst time improves performance metrics. The program correctly calculated waiting and turnaround times by executing the shortest processes first, reducing the average waiting time compared to FCFS. However, SJF may not be practical in real-time systems unless the burst time is known beforehand. This experiment helped understand how job prioritization impacts scheduling efficiency and reinforced key concepts in non-preemptive scheduling.

## Lab-5

## OBJECTIVE:

- To implement the FIFO page replacement algorithm for managing memory pages in a virtual memory system.
- To understand how pages are loaded and replaced in memory using the FIFO strategy.

## THEORY:

A computer's main memory (RAM) is divided into frames, which are physical blocks of a fixed size. The operating system's memory management unit (MMU) handles the mapping of logical pages (from a process's virtual address space) to these physical frames. When a process tries to access a page that is not currently in a frame (a page fault), the operating system must choose a page to remove from memory to make room for the new one. The FIFO algorithm is one method for making this choice.

The FIFO algorithm operates on a simple principle: the page that was brought into memory first is the first one to be evicted. This behavior is analogous to a queue or a waiting line: the person who has been waiting the longest is served next.

### Data Structures

To implement the FIFO algorithm, you need a way to track the order in which pages entered memory. A **queue** is the most suitable data structure for this purpose.

- **Queue:** A queue maintains a chronological order of pages. When a page is loaded into memory, its page number is added to the rear (end) of the queue. When a page needs to be replaced, the page number at the front (head) of the queue is selected for removal.
- **Frame Array/List:** This represents the physical memory frames. It stores the actual page numbers currently residing in memory.
- **Hash Map/Set:** To quickly check if a page is already in a frame, a hash map or a set can be used. The key would be the page number, allowing for efficient O(1) average-time lookups.

### Algorithm Steps

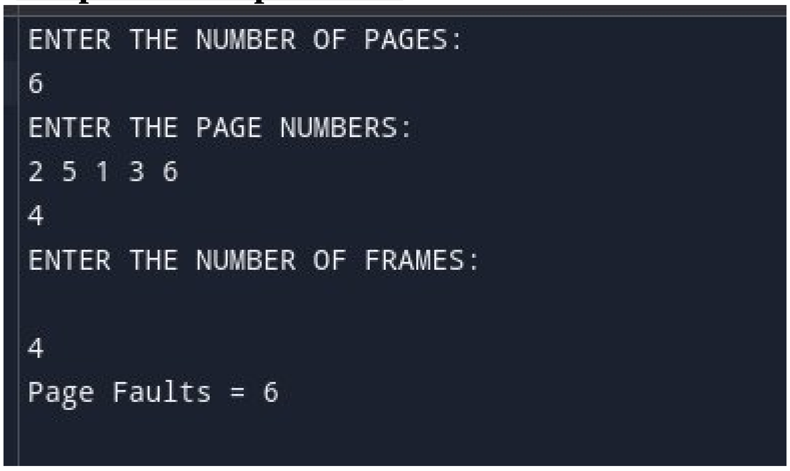
The implementation of the FIFO algorithm can be broken down into the following steps:

1. **Initialize** Start with an empty queue and an empty set of pages in memory.
2. **Page Reference:** When a page reference request arrives, check if the page is already in memory using the hash set.
3. **Page Hit:** If the page is found (a hit), do nothing. No page fault occurs, and the queue remains unchanged.
4. **Page Fault:** If the page is not found (a fault), follow these steps:
    - **Case 1: Empty Frame Available:** If there is an empty frame, load the new page into it. Add the page number to the end of the queue and the hash set.
    - **Case 2: No Empty Frame:** If all frames are full, a page must be replaced.
        - Take the page number from the front of the queue. This is the victim page.
        - Remove the victim page from the hash set.
        - Remove the victim page from the front of the queue.
        - Load the new page into the frame previously occupied by the victim page.
        - Add the new page number to the end of the queue and the hash set.

**Source Code:**

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

int pagefault(int a[], int frame[], int n, int no)
{
    int i, j, avail, count = 0, k; for (i
    = 0; i < no; i++) {
        frame[i] = -1;
    }
    j = 0;
    for (i = 0; i < n; i++) { avail
        = 0;
        for (k = 0; k < no; k++) { if
            (frame[k] == a[i]) {
                avail = 1;
                break;
            }
        }
        if (avail == 0)
            { frame[j] = a[i]; j
            = (j + 1) % no;
            count++;
        }
    }
    return count;
}

int main()
{
    int n, i, *a, *frame, no, fault;
    printf("ENTER THE NUMBER OF PAGES:\n");
    scanf("%d", &n);
    a = (int *)malloc(n * sizeof(int)); printf("ENTER
    THE PAGE NUMBERS:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("ENTER THE NUMBER OF FRAMES:\n");
    scanf("%d", &no);
    frame = (int *)malloc(no * sizeof(int)); fault
    = pagefault(a, frame, n, no); printf("Page
    Faults = %d\n", fault); free(a);
    free(frame); return
    0;
}
```

## Output and Explanation:


```
ENTER THE NUMBER OF PAGES:
6
ENTER THE PAGE NUMBERS:
2 5 1 3 6
4
ENTER THE NUMBER OF FRAMES:

4
Page Faults = 6
```

This C program implements the FIFO page replacement algorithm to calculate page faults in memory management. It takes input for the number of pages, the page reference string, and the number of frames. Pages are inserted into frames sequentially, and if a page is not present, a page fault occurs. When frames are full, the oldest page is replaced in a circular manner using modulo arithmetic. Finally, the program outputs the total number of page faults. It demonstrates fundamental operating system concepts practically.

## CONCLUSION:

The implementation of the FIFO page replacement algorithm in this lab successfully demonstrates the working principle of handling page faults in memory management. By maintaining a simple queue-based order, the algorithm ensures that the oldest loaded page is replaced first whenever a new page is required and memory is full. This approach, though straightforward, helps students understand how operating systems manage memory using replacement strategies. The program effectively simulates page faults, reinforcing theoretical concepts with practical execution. Through this experiment, we gain clear insights into FIFO’s strengths and limitations, strengthening understanding of fundamental operating system mechanisms.

## Lab-6:

**OBJECTIVE:** To implement the LRU Page Replacement Algorithm using a suitable programming language and evaluate the number of page faults that occur for a given sequence of pages and memory capacity.

## THEORY:

The Least Recently Used (LRU) page replacement algorithm is a memory management technique used in operating systems. Its fundamental theory is to replace the page that has not been used for the longest period of time. The logic behind this is that a page that has been referenced recently is more likely to be referenced again in the near future. This principle is based on the concept of temporal locality, a key characteristic of program execution where a process tends to reuse the same data and instructions that it has accessed recently.Unlike the simple FIFO (First-In, First-Out) algorithm, which suffers from Belady's Anomaly, LRU is a stack algorithm, meaning it will not exhibit this anomaly. Increasing the number of memory frames will always either decrease or keep the number of page faults the same.

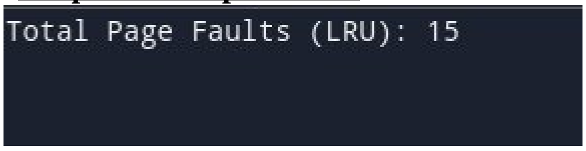
<u>Advantages and Disadvantages:</u>

- Advantages**:** LRU is a very effective and widely used algorithm. Its performance is generally superior to FIFO because it makes eviction decisions based on past usage patterns, which are a good predictor of future use. It also does not suffer from Belady's Anomaly.
- Disadvantages**:** Implementing LRU is more complex and computationally intensive than FIFO. It requires hardware or software to keep track of the recency of each page, which can be expensive. For a large number of pages, managing the doubly linked list and hash map can add overhead. While a doubly linked list provides an efficient implementation, other LRU variants exist approximate its behavior with less overhead, such as the Clock Algorithm or second-chance algorithms.

## Source Code:

```
#include <stdio.h> #include
<limits.h>
int lruPageReplacement(int *referenceString, int numPages, int numFrames) { int
    frames[numFrames];
    int lastUsed[numFrames]; int
    pageFaults = 0;
    int currentTime = 0;
    for (int i = 0; i < numFrames; i++) { frames[i] = -
        1;
        lastUsed[i] = 0;
    }
    for (int i = 0; i < numPages; i++) { int
        page = referenceString[i];
        int found = 0;
        for (int j = 0; j < numFrames; j++) { if
            (frames[j] == page) {
                found = 1;
                lastUsed[j] = currentTime; break;
            }
        }
        if (!found)
            { pageFaults++;
            int replaceIndex = 0;
            int minTime = INT_MAX;
            for (int j = 0; j < numFrames; j++) { if
                (lastUsed[j] < minTime) {
                    minTime = lastUsed[j]; replaceIndex
                    = j;
                }
            }
            frames[replaceIndex] = page;
            lastUsed[replaceIndex] = currentTime;
        }
        currentTime++;
    }
    return pageFaults;
}

int main() {
    int referenceString[] = {1, 2, 3, 4, 2, 1, 5, 2, 3, 4, 6, 2, 1, 3, 7, 2, 3, 4};
    int numPages = sizeof(referenceString) / sizeof(referenceString[0]); int
    numFrames = 3;
    int pageFaults = lruPageReplacement(referenceString, numPages, numFrames); printf("Total Page
    Faults (LRU): %d\n", pageFaults);
    return 0;
}
```

**Output and Explanation:**



```
Total Page Faults (LRU): 15
```

This C program implements the Least Recently Used (LRU) page replacement algorithm. It begins with a reference string of pages and a fixed number of frames. Two arrays are used: one to store the pages currently in memory and another to track when each page was last used. For each page reference, the program checks if it exists in memory. If not, a page fault occurs, and the least recently used page is replaced using the smallest last-used timestamp. The process continues until all references are processed, and finally, the program outputs the total number of page faults.

## CONCLUSION:

The implementation of the LRU page replacement algorithm successfully demonstrates efficient memory management by replacing the least recently used pages. Unlike FIFO, LRU avoids Belady’s Anomaly and provides better accuracy by utilizing temporal locality. This lab enhances understanding of practical paging mechanisms, page fault analysis, and operating system efficiency.

# Lab-7:

## OBJECTIVE:

- To understand the logic and performance of the First-Come, First-Served (FCFS) disk scheduling algorithm.
- To implement FCFS in C using sequential disk request handling and analyze seek time outcomes.

## THEORY:

The **First-Come, First-Served (FCFS)** disk-scheduling algorithm is the simplest approach to managing I/O requests for a hard disk. Its core theory dictates that disk requests are serviced strictly in the order they arrive in the disk queue. This is analogous to a typical queue at a supermarket, where the person who gets in line first is served first, regardless of what they are buying or how long it takes.

### Implementation

Implementing FCFS is straightforward. A queue data structure is used to hold the pending disk I/O requests. When a process needs to read or write data to the disk, its request (containing the cylinder/track number) is added to the rear of this queue. The disk arm, starting from its current position, then services the request at the front of the queue. Once that request is completed, the arm moves to service the next request at the front, and so on, until the queue is empty. There is no reordering or prioritization; requests are handled precisely in their arrival sequence.

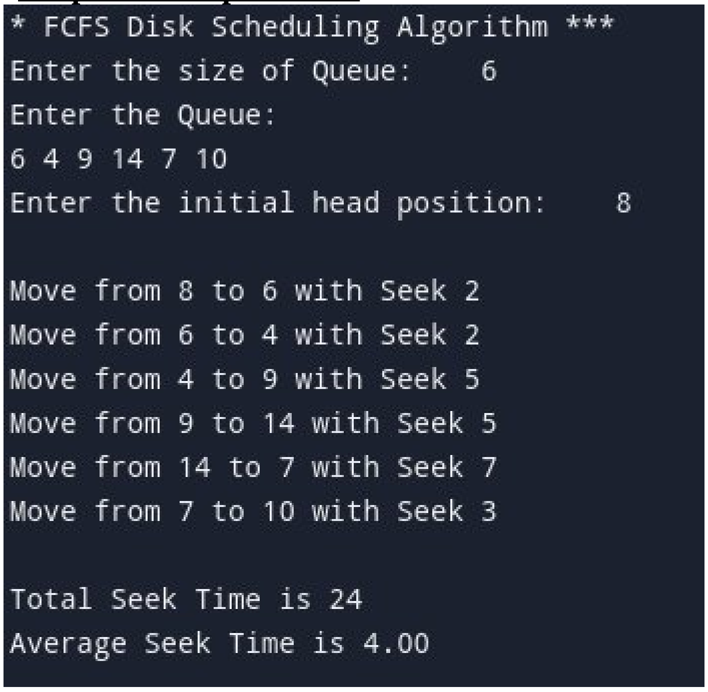
### Characteristics

FCFS is easy to understand and implement, making it inherently fair because every request eventually is serviced without discrimination. However, its simplicity often comes at a cost. It does not consider the physical location of the requests on the disk. This can lead to excessive head movement if consecutively arriving requests are far apart on the disk, resulting in long seek times and overall inefficient performance. For instance, if requests for cylinders 10, 100, and 11 arrive, the head will travel from 10 to 100, then all the way back to 11, instead of a more optimized path. Due to this potential for high seek times, FCFS is rarely used as the sole disk-scheduling algorithm in modern operating systems but serves as a baseline for comparison with algorithms that are more sophisticated.

## Source Code:

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

int main() {
    int queue[100], n, head, i, j, seek = 0, diff; float
    avg;
    printf("* FCFS Disk Scheduling Algorithm ***\n"); printf("Enter the
    size of Queue:\t");
    scanf("%d", &n); printf("Enter
    the Queue:\n"); for (i = 1; i <= n;
    i++) {
        scanf("%d", &queue[i]);
    }
    printf("Enter the initial head position:\t");
    scanf("%d", &head);
    queue[0] = head;
    printf("\n");
    for (j = 0; j < n; j++) {
        diff = abs(queue[j + 1] - queue[j]); seek
        += diff;
        printf("Move from %d to %d with Seek %d\n", queue[j], queue[j + 1], diff);
    }
    printf("\nTotal Seek Time is %d\n", seek); avg =
    (float)seek / n;
    printf("Average Seek Time is %.2f\n", avg);
    return 0;
}
```

**Output and Explanation:**



```
* FCFS Disk Scheduling Algorithm ***
Enter the size of Queue:    6
Enter the Queue:
6 4 9 14 7 10
Enter the initial head position:    8

Move from 8 to 6 with Seek 2
Move from 6 to 4 with Seek 2
Move from 4 to 9 with Seek 5
Move from 9 to 14 with Seek 5
Move from 14 to 7 with Seek 7
Move from 7 to 10 with Seek 3

Total Seek Time is 24
Average Seek Time is 4.00
```

The FCFS Disk Scheduling algorithm is one of the simplest approaches to manage disk access requests. It services requests in the order they arrive, similar to a queue system. This program calculates the total seek time by summing the absolute differences between consecutive disk positions and then derives the average seek time. Although easy to implement and understand, FCFS may lead to higher seek times compared to optimized algorithms, especially when requests are far apart on the disk.

## CONCLUSION:

Based on the implementation and analysis, the FCFS disk-scheduling algorithm is a foundational concept in operating systems, prioritizing simplicity and fairness by processing disk requests in the order they are received. The lab successfully demonstrated this sequential request handling, highlighting its ease of implementation. However, the results confirmed that FCFS is not optimal in terms of performance, often leading to a significant amount of disk arm movement and high seek times. This inefficiency makes it a less suitable choice for real-world scenarios but a crucial baseline for comparing more advanced disk scheduling algorithms.

## Lab-8

## OBJECTIVE

To implement the Optimal Page Replacement Algorithm in C and calculate the number of page faults for a given reference string and frame size.

## THEORY

In operating systems, memory management plays a crucial role in ensuring efficient execution of programs. Paging divides logical memory into pages and physical memory into frames. When a program requests a page not available in memory, a page fault occurs. If all frames are occupied, the system must decide which page to replace, and page replacement algorithms handle this.

The Optimal Page Replacement Algorithm, also known as OPT or MIN, is a theoretical technique designed to minimize the number of page faults. Its strategy is based on perfect knowledge of future memory references. Whenever a replacement is needed, it selects the page that will not be used for the longest duration in the future. By doing this, the algorithm guarantees the lowest possible number of page faults for a given sequence of references and frame allocation.

**Implementation follows a systematic process:**

1. If the required page is already in memory, no replacement is required.
2. If space is available, the new page is simply loaded.
3. If memory is full, the algorithm scans ahead in the reference string and replaces the page whose next use occurs farthest away or never reappears.

This method provides an ideal benchmark for evaluating practical algorithms like FIFO and LRU.

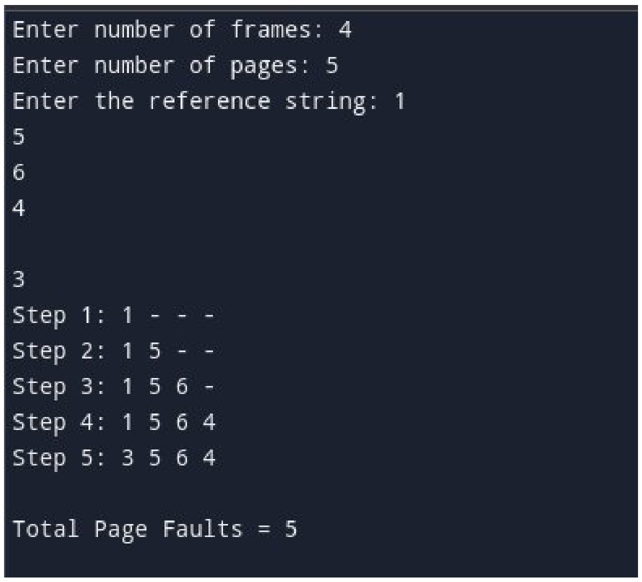
## Program Demonstration:

***<u>Source Code:</u>***

```
#include <stdio.h>

int findFarthest(int pages[], int memory[], int n, int index, int frames) { int
    farthest = index, pos = -1, found, i, j;
    for (i = 0; i < frames; i++) { found =
        0;
        for (j = index; j < n; j++) {
            if (memory[i] == pages[j]) { if
                (j > farthest) {
                    farthest = j; pos
                    = i;
                }
                found = 1; break;
            }
        }
        if (!found) return i;
    }
    return (pos == -1) ? 0 : pos;
}
int main() {
    int frames, n, i, j, page_faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &frames); printf("Enter
    number of pages: "); scanf("%d", &n);
    int pages[n], memory[frames];
    printf("Enter the reference string: "); for (i
    = 0; i < n; i++)
        scanf("%d", &pages[i]); for (i
    = 0; i < frames; i++)
        memory[i] = -1;
    for (i = 0; i < n; i++) { int
        page   =   pages[i];   int
        found = 0;
        for (j = 0; j < frames; j++) { if
            (memory[j] == page) {
                found  =  1;
                break;
            }
        }
         if (!found) {
              int placed = 0;
              for (j = 0; j < frames; j++) { if
            (memory[j] == -1) { memory[j]
                              = page;
                    placed = 1;
                    break;
                }
            }
            if (!placed) {
                int pos = findFarthest(pages, memory, n, i + 1, frames); memory[pos]
                = page;
            }
            page_faults++;
}
        printf("Step %d: ", i + 1); for (j
        = 0; j < frames; j++) {
            if (memory[j] != -1)
                printf("%d ", memory[j]);
            else
                printf("- ");
        }
```

```
printf("\n");
        }
        printf("\nTotal Page Faults = %d\n", page_faults);
        return 0;
    }
```

## Output:



```
Enter number of frames: 4
Enter number of pages: 5
Enter the reference string: 1
5
6
4

3
Step 1: 1 - - -
Step 2: 1 5 - -
Step 3: 1 5 6 -
Step 4: 1 5 6 4
Step 5: 3 5 6 4

Total Page Faults = 5
```

## CONCLUSION:

The implementation of the Optimal Page Replacement Algorithm demonstrates how efficient memory management can minimize page faults in paging systems. By predicting future references and replacing the page that will not be used for the longest period, the algorithm achieves the best possible performance. Although it is impractical in real-world applications due to the requirement of future knowledge, it serves as a valuable benchmark for evaluating practical algorithms like FIFO and LRU in operating system design.

## Lab-9

## OBJECTIVE:

To implement the Banker’s Algorithm in C for deadlock avoidance by determining whether a given system state is safe, ensuring that resource allocation decisions prevent the possibility of deadlock while maintaining system stability.

## THEORY:

The Banker's Algorithm is a deadlock avoidance algorithm used in operating systems to ensure that all processes can complete their execution without encountering a deadlock. It works by carefully checking the system's state before granting resource requests, ensuring the system remains in a safe state. The core principle behind the Banker's Algorithm is to simulate resource allocation before actually granting a request. It is like a careful banker who only approves a loan if they are certain they can cover all outstanding liabilities and future requests, preventing bankruptcy (deadlock, in this case).

The algorithm requires each process to declare its maximum resource needs upfront. Based on this, it maintains several data structures:

a) Available: A vector indicating the number of available instances of each resource type. This is the pool of resources currently free for allocation.

b) Max: A matrix defining the maximum number of instances of each resource type that each process might ever request.

c) Allocation: A matrix showing the number of instances of each resource type currently allocated to each process.

d) Need: A matrix representing the remaining resources each process still requires completing its execution. This is calculated as Need[i][j] = Max[i][j] - Allocation[i][j].

## Program Demonstration:

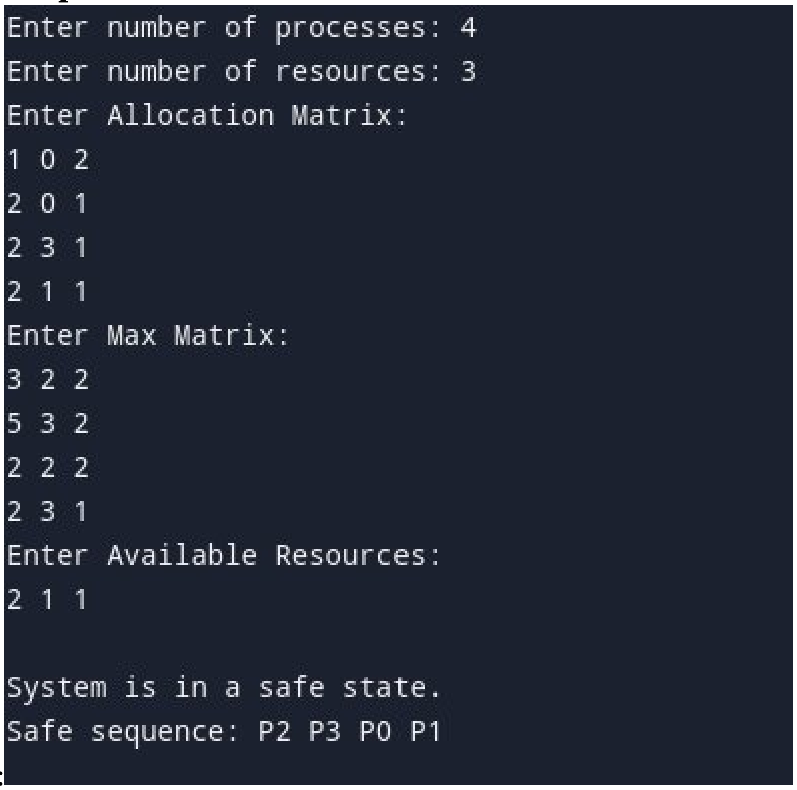
***Source Code:***

```
#include <stdio.h>

int main() {
    int n, m, i, j, k;
    printf("Enter number of processes: ");
    scanf("%d", &n);
```

```
        printf("Enter number of resources: ");
        scanf("%d", &m);

        int alloc[n][m], max[n][m], avail[m];
        printf("Enter Allocation Matrix:\n"); for (i
        = 0; i < n; i++)
            for (j = 0; j < m; j++)
                scanf("%d", &alloc[i][j]);

        printf("Enter Max Matrix:\n"); for
        (i = 0; i < n; i++)
            for (j = 0; j < m; j++) scanf("%d",
                &max[i][j]);

        printf("Enter Available Resources:\n"); for (i
        = 0; i < m; i++)
            scanf("%d", &avail[i]);

        int need[n][m];
        for (i = 0; i < n; i++) for (j
            = 0; j < m; j++)
                need[i][j] = max[i][j] - alloc[i][j];

        int finish[n], safeSeq[n], index = 0; for
        (i = 0; i < n; i++)
            finish[i] = 0;

        int work[m];
        for (i = 0; i < m; i++)
            work[i] = avail[i];
        while (index < n) { int
            found = 0;
            for (i = 0; i < n; i++) { if
                (finish[i] == 0) {
                    int canAllocate = 1;
                    for (j = 0; j < m; j++) {
                        if (need[i][j] > work[j])
                            { canAllocate = 0; break;
                        }
                    }
                    if (canAllocate) {
                        for (k = 0; k < m; k++) work[k]
                            += alloc[i][k];
safeSeq[index++] = i; finish[i] = 1;
                        found = 1;
                    }
                }
            }
            if (!found) {
```

```
        printf("\nSystem is in an unsafe state.
Deadlock may occur.\n");
            return 0;
        }
    }
    printf("\nSystem is in a safe state.\nSafe sequence: ");
    for (i = 0; i < n; i++)
        printf("P%d ", safeSeq[i]);
    printf("\n"); return
    0;
}
```

**Output**



```
Enter number of processes: 4
Enter number of resources: 3
Enter Allocation Matrix:
1 0 2
2 0 1
2 3 1
2 1 1
Enter Max Matrix:
3 2 2
5 3 2
2 2 2
2 3 1
Enter Available Resources:
2 1 1

System is in a safe state.
Safe sequence: P2 P3 P0 P1
```

:

## CONCLUSION:

The implementation of the Banker’s Algorithm confirms its effectiveness in avoiding deadlocks by ensuring system safety before resource allocation. By calculating the need matrix and simulating allocations, it determines a safe sequence if one exists. This approach guarantees reliable resource management and prevents unsafe states in operating systems.

## Lab-10

## OBJECTIVE:

- To avoid deadlock and starvation among concurrent processes**.**
- To implement the Dining Philosophers Problem in C using semaphores.

## THEORY:

The Dining Philosophers Problem, proposed by Dijkstra, is a well-known synchronization challenge that demonstrates issues in resource sharing among parallel processes. Five philosophers sit at a round table, alternating between eating and thinking. A single fork is placed between each pair, and a philosopher requires both the left and right fork to dine.

This problem emphasizes the importance of synchronization to achieve:

- Mutual exclusion: Ensuring that no two philosophers use the same fork at once.
- Deadlock prevention**:** Avoiding situations where all philosophers are stuck waiting forever.
- Starvation prevention: Guaranteeing that every philosopher eventually gets to eat.

Semaphores are applied to manage fork access, allowing philosophers to eat safely while avoiding deadlock and starvation.

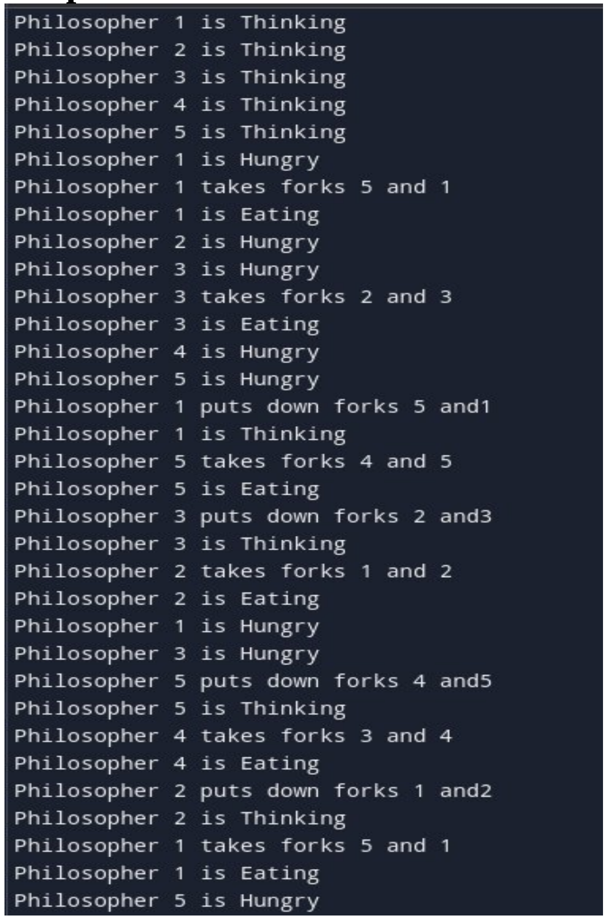
### Program Demonstration:

**Source Code:**

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
sem_t mutex;
sem_t S[N];
int state[N];
int phil[N] = {0, 1, 2, 3, 4};

void test(int i) {

if (state[i] == HUNGRY &&

        state[(i + 4) % N] != EATING &&
        state[(i + 1) % N] != EATING) {
        state[i] = EATING;
        sleep(1);
        printf("Philosopher %d takes forks %d and %d\n", i
+ 1, (i + 4) % N + 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]);
        sleep(1);
    }

    void put_fork(int i)
        { sem_wait(&mutex);
        state[i] = THINKING;
        printf("Philosopher %d puts down forks %d and
    %d\n", i + 1, (i + 4) % N + 1, i + 1);
        printf("Philosopher %d is Thinking\n", i + 1); test((i +
        4) % N);
        test((i + 1) % N);
        sem_post(&mutex);
    }

    void* philosopher(void* num) { while
        (1) {
            int i = *(int*)num;
            sleep(1); take_fork(i);
            sleep(2); 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:**



```
Philosopher 1 is Thinking
Philosopher 2 is Thinking
Philosopher 3 is Thinking
Philosopher 4 is Thinking
Philosopher 5 is Thinking
Philosopher 1 is Hungry
Philosopher 1 takes forks 5 and 1
Philosopher 1 is Eating
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes forks 2 and 3
Philosopher 3 is Eating
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 1 puts down forks 5 and1
Philosopher 1 is Thinking
Philosopher 5 takes forks 4 and 5
Philosopher 5 is Eating
Philosopher 3 puts down forks 2 and3
Philosopher 3 is Thinking
Philosopher 2 takes forks 1 and 2
Philosopher 2 is Eating
Philosopher 1 is Hungry
Philosopher 3 is Hungry
Philosopher 5 puts down forks 4 and5
Philosopher 5 is Thinking
Philosopher 4 takes forks 3 and 4
Philosopher 4 is Eating
Philosopher 2 puts down forks 1 and2
Philosopher 2 is Thinking
Philosopher 1 takes forks 5 and 1
Philosopher 1 is Eating
Philosopher 5 is Hungry
```

The code implements the Dining Philosophers Problem using threads and semaphores. Each philosopher can be THINKING, HUNGRY, or EATING, stored in the `state` array. A mutex semaphore ensures mutual exclusion when accessing shared resources, while individual semaphores control when philosophers can pick up forks. The `take_fork` and `put_fork` functions manage fork allocation, and the `test` function checks neighbors before allowing eating. This design prevents deadlock and starvation, ensuring that all philosophers eventually eat while safely sharing forks. Threads simulate concurrent philosopher actions.

## CONCLUSION:

In this lab, the Dining Philosophers Problem was successfully implemented in C using threads and semaphores to demonstrate proper synchronization among concurrent processes. The program effectively modeled five philosophers alternating between thinking and eating, with forks represented as shared resources. By using a mutex semaphore for mutual exclusion and individual semaphores for each philosopher, the program ensured safe access to forks, preventing simultaneous use by neighboring philosophers. The `take_fork` and `put_fork` functions, along with the `test` function, provided a controlled mechanism for resource allocation while continuously monitoring the state of each philosopher. This approach avoided deadlock, where all philosophers could potentially wait indefinitely, and starvation, ensuring that every philosopher eventually got a chance to eat. The program also illustrated the importance of synchronization in concurrent systems, showing how careful management of shared resources allows multiple processes to operate safely without conflict. Overall, the lab reinforced key concepts of mutual exclusion, deadlock prevention, and starvation avoidance in operating system design.