

EXP NO: 11

REG.NO:

DATE:

NAME:

INTERPROCESS COMMUNICATION USING PIPES

AIM:

To implement IPC using pipes in C program.

PROGRAM DESCRIPTION:

In IPC (Inter-Process Communication) program, pipes are used to establish communication between multiple processes. The program simulates a message distribution system where a parent process sends messages on different topics to two audiences through a separator process. The parent process generates a set of messages, each tagged with a topic ("Technology" or "Health"), and sends them through a main pipe.

- **Separator Process:** This process reads messages from the main pipe, checks each message's topic, and redirects it to the appropriate audience pipe based on the topic. Messages with the topic "Technology" are sent to the audience for "Technology," while messages on "Health" are directed to the respective audience.
- **Audience Processes:** Two audience processes (one for each topic) read from their assigned pipes. When they receive messages matching their topic, they display the message content.

HEADER FILES USED :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <wait.h>
```

PROGRAM CODE:

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

```
#include <string.h>
#include <time.h>
#include <wait.h>

#define NUM_MESSAGES 10
#define TOPIC_A "Technology"
#define TOPIC_B "Health"

const char *topic_a_contents[] = {
    "AI is transforming industries.",
    "Quantum computing is the future.",
    "Blockchain technology is revolutionizing finance."
};

const char *topic_b_contents[] = {
    "Eating healthy boosts immunity.",
    "Regular exercise improves mental health.",
    "Sleep is crucial for well-being."
};

typedef struct {
    char topic[20];
    char content[100];
} Message;

void audience(const char *topic, int read_fd) {
    Message msg;
    while (read(read_fd, &msg, sizeof(msg)) > 0) {
        if (strcmp(msg.topic, topic) == 0) {
            printf("%s Audience received: %s\n", topic, msg.content);
        }
    }
}
```

```

    }
}
close(read_fd);
exit(0);
}

```

```

int main() {
    int pipe_main[2], pipe_a[2], pipe_b[2];

    // Create pipes
    pipe(pipe_main);
    pipe(pipe_a);
    pipe(pipe_b);

    // Fork separator process
    if (fork() == 0) { // Separator
        close(pipe_main[1]);
        close(pipe_a[0]);
        close(pipe_b[0]);

        Message msg;
        while (read(pipe_main[0], &msg, sizeof(msg)) > 0) {
            if (strcmp(msg.topic, TOPIC_A) == 0) {
                write(pipe_a[1], &msg, sizeof(msg));
            } else {
                write(pipe_b[1], &msg, sizeof(msg));
            }
        }
        close(pipe_a[1]);
        close(pipe_b[1]);
    }
}

```

```

        exit(0);
    }

// Fork audience for Topic A
if (fork() == 0) {
    close(pipe_main[0]);
    close(pipe_a[1]);
    audience(TOPIC_A, pipe_a[0]);
}

// Fork audience for Topic B
if (fork() == 0) {
    close(pipe_main[0]);
    close(pipe_b[1]);
    audience(TOPIC_B, pipe_b[0]);
}

// Parent process
close(pipe_main[0]);
srand(time(NULL));
for (int i = 0; i < NUM_MESSAGES; i++) {
    Message msg;
    if (rand() % 2 == 0) {
        strcpy(msg.topic, TOPIC_A);
        strcpy(msg.content, topic_a_contents[rand() % 3]);
    } else {
        strcpy(msg.topic, TOPIC_B);
        strcpy(msg.content, topic_b_contents[rand() % 3]);
    }
    write(pipe_main[1], &msg, sizeof(msg));
}

```

```

    printf("Parent sent: [%s] %s\n", msg.topic, msg.content);
    sleep(1);
}

close(pipe_main[1]);
wait(NULL); // Wait for separator
wait(NULL); // Wait for audience A
wait(NULL); // Wait for audience B

return 0;
}

```

SAMPLE INPUT AND OUTPUT:

```

Parent sent: [Technology] AI is transforming industries.
Technology Audience received: AI is transforming industries.
Parent sent: [Health] Regular exercise improves mental health.
Health Audience received: Regular exercise improves mental health.
Parent sent: [Technology] Blockchain technology is revolutionizing finance.
Technology Audience received: Blockchain technology is revolutionizing finance.
Parent sent: [Technology] AI is transforming industries.
Technology Audience received: AI is transforming industries.
Parent sent: [Health] Sleep is crucial for well-being.
Health Audience received: Sleep is crucial for well-being.
Parent sent: [Health] Regular exercise improves mental health.
Health Audience received: Regular exercise improves mental health.
Parent sent: [Technology] Quantum computing is the future.
Technology Audience received: Quantum computing is the future.
Parent sent: [Health] Regular exercise improves mental health.
Health Audience received: Regular exercise improves mental health.
Parent sent: [Technology] Quantum computing is the future.
Technology Audience received: Quantum computing is the future.
Parent sent: [Health] Eating healthy boosts immunity.
Health Audience received: Eating healthy boosts immunity.

```

RESULT:

Thus, the IPC using pipes has been implemented successfully using C program.

EXP NO: 12

REG NO:

DATE:

NAME:

INTERPROCESS COMMUNICATION USING SHARED MEMORY

AIM:

The aim of this program is to implement shared memory using IPC in C.

ALGORITHM:

Algorithm for `server.c` (Server Process)

1. Create a shared memory segment using a unique key and specify the size.
2. Attach to the shared memory segment.
3. Print a message indicating that the server is waiting for input from the client.
4. Enter an infinite loop:
 - Continuously check if the client has written data to the shared memory.
 - If data is available, convert the received Celsius value to a floating-point number.
 - Convert the Celsius temperature to Fahrenheit.
 - Write the converted Fahrenheit value back to the shared memory.
 - Print a message on the server indicating the conversion.
 - Clear the shared memory to indicate it is ready for the next input.
 - Wait briefly to reduce CPU usage.
5. Optionally, detach from the shared memory when the server exits..

Algorithm for `client.c` (Client Process)

1. Access the existing shared memory segment using the same key as the server.
2. Attach to the shared memory segment.

3. Enter a loop to take user input:

- Prompt the user to enter a temperature in Celsius.
 - If the user enters '-999', break the loop and exit.
 - Write the entered Celsius value to the shared memory.
 - Wait briefly to allow the server to process the input.
 - Read and display the converted Fahrenheit temperature from the shared memory.
 - Clear the shared memory to prepare for the next input.
4. Detach from the shared memory when the client exits.

PROGRAM CODE:

SERVER.C

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

#define SHM_SIZE 128 // Define shared memory size

int main() {
    int shmid;
    key_t key = 1234; // Unique key for the shared memory
    char *data;

    // Create shared memory segment
    shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("shmget failed");
```

```

    exit(1);
}

// Attach to shared memory
data = (char *)shmat(shmid, NULL, 0);
if (data == (char *)(-1)) {
    perror("shmat failed");
    exit(1);
}

// Print message once and wait until input is provided
printf("Waiting for temperature input (Celsius) from client...\n");

while (1) {
    // Check if client has written to shared memory
    if (data[0] != '\0') {
        float celsius = atof(data);
        float fahrenheit = (celsius * 9 / 5) + 32;
        sprintf(data, "Fahrenheit: %.2f", fahrenheit);
        printf("Converted %.2f Celsius to %.2f Fahrenheit\n", celsius, fahrenheit);
        data[0] = '\0'; // Reset shared memory for next input

        // Print waiting message again after processing the input
        printf("Waiting for temperature input (Celsius) from client...\n");
    }
    usleep(100000); // Short wait to reduce CPU usage
}

// Detach from shared memory
shmdt(data);

```



```
    return 0;
}
```

CLIENT.C

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

#define SHM_SIZE 128

int main() {
    int shmid;
    key_t key = 1234; // Same key as server
    char *data;

    // Access shared memory segment
    shmid = shmget(key, SHM_SIZE, 0666);
    if (shmid == -1) {
        perror("shmget failed");
        exit(1);
    }

    // Attach to shared memory
    data = (char *)shmat(shmid, NULL, 0);
    if (data == (char *)(-1)) {
        perror("shmat failed");
        exit(1);
    }
}
```

```
while (1) {  
    // Get temperature input from user  
    float celsius;  
    printf("Enter temperature in Celsius (or -999 to exit): ");  
    scanf("%f", &celsius);  
  
    if (celsius == -999) {  
        break; // Exit condition  
    }  
    // Write to shared memory  
    sprintf(data, "%.2f", celsius);  
    // Wait for the server to process  
    sleep(2);  
    // Read the converted temperature  
    printf("%s\n", data);  
    // Reset shared memory for next input  
    data[0] = '\0';  
}  
// Detach from shared memory  
shmdt(data);  
return 0;  
}
```

TEST CASE:

SERVER.C

```
crysis@DESKTOP-5C80AA8:/mnt/c/Users/Admin/OneDrive/Desktop$  
gcc server.c -o server  
crysis@DESKTOP-5C80AA8:/mnt/c/Users/Admin/OneDrive/Desktop$  
./server  
Waiting for temperature input (Celsius) from client...  
Converted 35.00 Celsius to 95.00 Fahrenheit  
Waiting for temperature input (Celsius) from client...  
Converted 45.00 Celsius to 113.00 Fahrenheit  
Waiting for temperature input (Celsius) from client...  
Converted 55.00 Celsius to 131.00 Fahrenheit  
Waiting for temperature input (Celsius) from client...  
^C  
crysis@DESKTOP-5C80AA8:/mnt/c/Users/Admin/OneDrive/Desktop$  
█
```

CLIENT.C

```
crysis@DESKTOP-5C80AA8:/mnt/c/Users/Admin/OneDrive/Desktop$  
gcc client.c -o client  
crysis@DESKTOP-5C80AA8:/mnt/c/Users/Admin/OneDrive/Desktop$  
./client  
Enter temperature in Celsius (or -999 to exit): 35  
  
Enter temperature in Celsius (or -999 to exit): 45  
  
Enter temperature in Celsius (or -999 to exit): 55  
  
Enter temperature in Celsius (or -999 to exit): -999  
crysis@DESKTOP-5C80AA8:/mnt/c/Users/Admin/OneDrive/Desktop$  
█
```

RESULT:

Thus, the IPC using shared memory has been executed successful using C program.

EXP NO: 13

REG NO:

DATE:

NAME:

SOLVING DINING PHILOSOPHER'S USING SEMAPHORES

AIM:

The aim of this program is to simulate the Dining Philosophers Problem using semaphores in C. The objective is to solve the synchronization issue where philosophers need forks to eat, without leading to deadlock.

PROBLEM DESCRIPTION:

Create a multithreaded program where five philosophers sit around a table and alternate between thinking and eating. Philosophers share forks placed between them, and no two neighbouring philosophers can use the same fork simultaneously. Use POSIX threads and semaphores to prevent deadlock and ensure synchronization.

LIBRARY FUNCTIONS USED:

- `pthread_create()`
- `pthread_join()`
- `sem_init()`
- `sem_wait()`
- `sem_post()`
- `usleep()`

EXPLANATION OF KEY FUNCTIONS:

1. **`pthread_create()`:**
This function creates a new thread that will execute a given function. It takes four parameters: a thread identifier, thread attributes, the function to be executed, and arguments to pass to that function.
2. **`pthread_join()`:**
This function makes the main thread wait for other threads to complete. It ensures that all threads finish execution before the program exits.

3. sem_init():

This function initializes a semaphore. It takes three parameters: the address of the semaphore, a flag (0 for local use), and the initial value of the semaphore.

4. sem_wait():

This function decrements the semaphore value. If the semaphore value is zero, it blocks the thread until the semaphore becomes available.

5. sem_post():

This function increments the semaphore value, signaling that the resource is available for use by other threads.

6. usleep():

This function suspends the execution of a thread for a specified number of microseconds.

PROGRAM CODE:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_PHILOSOPHERS 5
#define ITERATIONS 10 // Number of times each philosopher will think and eat
sem_t forks[NUM_PHILOSOPHERS]; // One semaphore per fork
sem_t mutex; // Semaphore to control the number of philosophers trying to eat

void think(int philosopher_number) {
    printf("Philosopher %d is thinking.\n", philosopher_number);
    usleep(rand() % 1000); // Random thinking time
}

void eat(int philosopher_number) {
    printf("Philosopher %d is eating.\n", philosopher_number);
```

```

        usleep(rand() % 1000); // Random eating time
    }

void* philosopher(void* num) {
    int philosopher_number = *(int*)num;

    for (int i = 0; i < ITERATIONS; i++) {
        think(philosopher_number);

        // only N-1 philosophers can pick up forks at the same time to avoid deadlock
        sem_wait(&mutex);

        // Pick up the forks
        int left_fork = philosopher_number;
        int right_fork = (philosopher_number + 1) % NUM_PHILOSOPHERS;

        sem_wait(&forks[left_fork]); // Pick up left fork
        printf("Philosopher %d picked up left fork.\n", philosopher_number);
        sem_wait(&forks[right_fork]); // Pick up right fork
        printf("Philosopher %d picked up right fork.\n", philosopher_number);

        eat(philosopher_number);

        // Put down the forks
        sem_post(&forks[left_fork]); // Put down left fork
        sem_post(&forks[right_fork]); // Put down right fork

        printf("Philosopher %d put down forks.\n", philosopher_number);

        // Allow another philosopher to try to eat
    }
}

```

```

        sem_post(&mutex);
    }

    printf("Philosopher %d is done.\n", philosopher_number);
    return NULL;
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_numbers[NUM_PHILOSOPHERS];

    // Initialize semaphores for forks and mutex
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&forks[i], 0, 1); // Each fork is initially available (1)
    }

    sem_init(&mutex, 0, NUM_PHILOSOPHERS - 1); // Allow up to N-1 philosophers to
    pick up forks

    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_numbers[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &philosopher_numbers[i]);
    }

    // Join philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Destroy semaphores

```

```
for (int i = 0; i < NUM_PHILOSOPHERS; i++) {  
    sem_destroy(&forks[i]);  
}  
sem_destroy(&mutex);  
return 0;  
}
```

OUTPUT:

```
● zia@kez:~$ gcc dining.c -o dining  
● zia@kez:~$ ./dining  
Philosopher 0 is thinking.  
Philosopher 1 is thinking.  
Philosopher 2 is thinking.  
Philosopher 3 is thinking.  
Philosopher 4 is thinking.  
Philosopher 0 picked up left fork.  
Philosopher 0 picked up right fork.  
Philosopher 0 is eating.  
Philosopher 2 picked up left fork.  
Philosopher 2 picked up right fork.  
Philosopher 2 is eating.  
Philosopher 1 picked up left fork.  
Philosopher 0 put down forks.  
Philosopher 0 is thinking.  
Philosopher 4 picked up left fork.  
Philosopher 4 picked up right fork.  
Philosopher 4 is eating.  
Philosopher 1 picked up right fork.  
Philosopher 3 picked up left fork.  
Philosopher 1 is eating.  
Philosopher 2 put down forks.  
Philosopher 2 is thinking.  
Philosopher 4 put down forks.  
Philosopher 4 is thinking.
```

RESULT:

Thus, Dining Philosophers Program has been implemented and run successfully.

EXP. NO: 14

REG.NO:

DATE:

NAME:

THREADING AND SYNCHRONIZATION APPLICATIONS: READERS-WRITERS PROBLEM

AIM:

To implement the reader-writer problem using threading and synchronization using a C program.

PROBLEM STATEMENT:

The Reader-Writer problem involves coordinating access to a shared resource where multiple readers can read concurrently, while writers require exclusive access. Effective synchronization is essential to prevent conflicts and ensure data consistency.

PROBLEM DESCRIPTION:

- **Definition:** Multiple readers can read a shared resource concurrently, while writers require exclusive access. Writers block reads and writes until they complete their operations.
- **Reader Behavior:** Readers increment a counter upon access, allowing multiple simultaneous reads. The first reader blocks writers, while the last reader releases them.
- **Writer Behavior:** Writers require exclusive access, preventing readers from accessing the resource while writing. They wait for their turn if another writer is active.
- **Synchronization Mechanism:** Semaphores manage access to the resource, and a mutex tracks the reader count. This ensures safe and conflict-free access to the shared resource.

STRUCTURE OF A PRODUCER-CONSUMER PROBLEM:

Writer

do {

 wait(rw_mutex);

```

        // Writing is performed

        signal(rw_mutex);
    } while (true);
# Reader
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);

    // Reading is performed

    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);

```

PROGRAM CODE:

```

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem_t wrt;

pthread_mutex_t mutex;

int read_count = 0;

int current_writer = 0; // To track the writer currently accessing the resource

```

```

int current_reader = 0; // To track the first reader accessing the resource
int writer_active = 0; // Flag to track if a writer is actually writing

void* writer(void* arg) {
    int writer_id = *((int*)arg);

    while (1) {
        printf("Writer %d is trying to write.\n", writer_id);

        // Now actually block until the writer can proceed
        sem_wait(&wrt);

        // Mark the current writer and set writer active flag
        current_writer = writer_id;
        writer_active = 1; // Writer is actively writing

        // Critical section (writing)
        printf("Writer %d is writing.\n", writer_id);
        sleep(1);

        // Writer finished
        printf("Writer %d finished writing. Unlocking the resource.\n", writer_id);
        writer_active = 0; // Writer finished writing
        current_writer = 0; // Reset current writer
        sem_post(&wrt); // Unlock resource

        // sleep(2);
    }
}

```

```

void* reader(void* arg) {
    int reader_id = *((int*)arg);

    while (1) {
        printf("Reader %d is trying to read.\n", reader_id);

        // Lock the mutex to update the reader count
        pthread_mutex_lock(&mutex);
        read_count++;
        if (read_count == 1) {
            // First reader blocks the writer and marks the current reader
            if (writer_active == 1) {
                printf("Reader %d waiting because resource is being written by Writer %d.\n",
                    reader_id, current_writer);
            }
            sem_wait(&wrt); // Lock the resource for readers
        }
        current_reader = reader_id;
        pthread_mutex_unlock(&mutex);

        // Critical section (reading)
        printf("Reader %d is reading.\n", reader_id);
        // sleep(1);

        // Lock the mutex to update the reader count
        pthread_mutex_lock(&mutex);
        read_count--;
        if (read_count == 0) {
            // Last reader releases the writer
            current_reader = 0; // Reset current reader
            printf("Reader %d finished reading. Unlocking the resource.\n", reader_id);
        }
    }
}

```

```

        sem_post(&wrt); // Unlock resource
    } else {
        printf("Reader %d finished reading.\n", reader_id);
    }
    pthread_mutex_unlock(&mutex);

    // sleep(2);
}
}

int main() {
    pthread_t readers[3], writers[2];
    int reader_ids[3] = {1, 2, 3};
    int writer_ids[2] = {1, 2};
    sem_init(&wrt, 0, 1);
    pthread_mutex_init(&mutex, NULL);

    // Create writer threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
    }

    // Create reader threads
    for (int i = 0; i < 3; i++) {
        pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
    }

    // Join threads (the program runs indefinitely, so this is just for demonstration)
    for (int i = 0; i < 2; i++) {
        pthread_join(writers[i], NULL);
    }
}

```

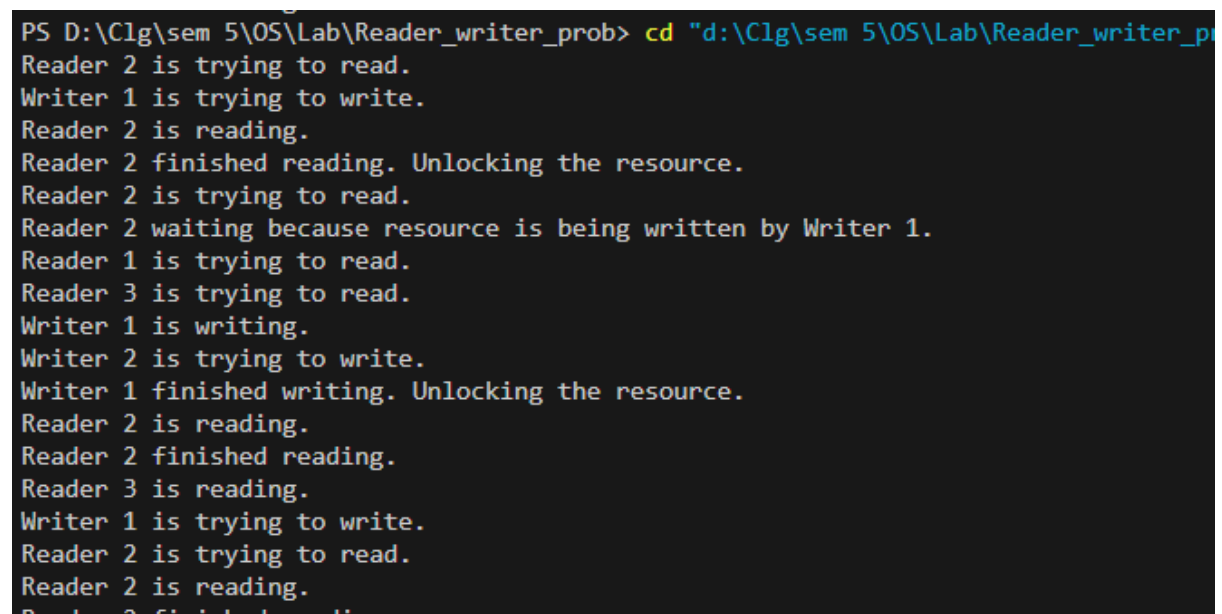
```

    }
    for (int i = 0; i < 3; i++) {
        pthread_join(readers[i], NULL);
    }
    sem_destroy(&wrt);
    pthread_mutex_destroy(&mutex);

    return 0;
}

```

SAMPLE INPUT AND OUTPUT:



```

PS D:\Clg\sem 5\OS\Lab\Reader_writer_prob> cd "d:\Clg\sem 5\OS\Lab\Reader_writer_prob"
Reader 2 is trying to read.
Writer 1 is trying to write.
Reader 2 is reading.
Reader 2 finished reading. Unlocking the resource.
Reader 2 is trying to read.
Reader 2 waiting because resource is being written by Writer 1.
Reader 1 is trying to read.
Reader 3 is trying to read.
Writer 1 is writing.
Writer 2 is trying to write.
Writer 1 finished writing. Unlocking the resource.
Reader 2 is reading.
Reader 2 finished reading.
Reader 3 is reading.
Writer 1 is trying to write.
Reader 2 is trying to read.
Reader 2 is reading.
Reader 2 finished reading.

```

RESULT:

Thus, the Reader-writer problem has been implemented using threading and synchronization

EXP NO: 15

REG.NO:

DATE:

NAME:

BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE

AIM:

To implement the Banker's Algorithm using a C program for deadlock avoidance in a system with multiple processes and resource types.

PROGRAM DESCRIPTION:

Banker's Algorithm is a resource allocation and deadlock avoidance algorithm that checks whether resources can be allocated to processes in a way that avoids a deadlock. It determines a **safe sequence** of execution, where each process can execute with available resources and no deadlock occurs.

CRITERIA TO BE CONSIDERED:

1. **Safe Sequence:** Ensures that processes execute in a sequence that does not lead to a deadlock.
2. **Need Matrix:** Calculated as $\text{Need} = \text{Max} - \text{Allocation}$ for each process.
3. **Available Resources:** Calculated as $\text{Available} = \text{Total} - \text{Sum of Allocated resources}$.
4. **Finish Flag:** Used to check if all processes have completed their execution.

HEADER FILES USED:

- `stdio.h`
-

PROGRAM CODE:

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

```
int main() {
```

```
    int p, r;
```

```

printf("Enter the number of processes : ");
scanf("%d", &p);

printf("Enter the number of resources : ");
scanf("%d", &r);


int allocation[p][r], max[p][r], need[p][r];
int available[r], finish[p], sum_alloc[r], ans[p], overall[r];


for (int j = 0; j < r; j++) {
    sum_alloc[j] = 0;
}


for (int j = 0; j < r; j++) {
    printf("Instances for resource R%d : ", j + 1);
    scanf("%d", &overall[j]);
}


for (int i = 0; i < p; i++) {
    for (int j = 0; j < r; j++) {
        printf("Allocated resources for R%d - P%d : ", j + 1, i);
        scanf("%d", &allocation[i][j]);
        sum_alloc[j] += allocation[i][j];
    }
    for (int j = 0; j < r; j++) {
        printf("Maximum resources for R%d - P%d : ", j + 1, i);
        scanf("%d", &max[i][j]);
    }
    finish[i] = 0;
}

```



```

for (int j = 0; j < r; j++) {
    available[j] = overall[j] - sum_alloc[j];
}

```

```

for (int i = 0; i < p; i++) {
    for (int j = 0; j < r; j++) {
        need[i][j] = max[i][j] - allocation[i][j];
    }
}

```

```

int ind = 0;
for (int k = 0; k < p; k++) {
    for (int i = 0; i < p; i++) {
        if (finish[i] == 0) {
            int flag = 0;
            for (int j = 0; j < r; j++) {
                if (need[i][j] > available[j]) {
                    flag = 1;
                    break;
                }
            }
            if (flag == 0) {
                ans[ind++] = i;
                for (int y = 0; y < r; y++)
                    available[y] += allocation[i][y];
                finish[i] = 1;
            }
        }
    }
}

```

```

int flag = 1;
for (int i = 0; i < p; i++) {
    if (finish[i] == 0) {
        flag = 0;
        printf("\nDeadlock cannot be avoided");
        break;
    }
}

if (flag == 1) {
    printf("\nSafe Sequence: ");
    for (int i = 0; i < p - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d", ans[p - 1]);
}

return 0;
}

```

SAMPLE INPUT AND OUTPUT:

1. Sample Input 1:

- Number of Processes: 3
- Number of Resources: 3
- Total Instances for Resources: R1 = 10, R2 = 5, R3 = 7
- Allocated Resources and Maximum Resources:

Allocated R1 for P0: 0

Allocated R2 for P0: 1

Allocated R3 for P0: 0

Max R1 for P0: 7

Max R2 for P0: 5

Max R3 for P0: 3

...

Output 1:

Safe Sequence: P0 -> P2 -> P1

2. Sample Input 2:

- Number of Processes: 5
- Number of Resources: 3
- Total Instances for Resources: R1 = 10, R2 = 5, R3 = 7
- Allocated Resources and Maximum Resources:

Allocated R1 for P0: 0

Allocated R2 for P0: 1

Allocated R3 for P0: 0

Max R1 for P0: 7

Max R2 for P0: 5

Max R3 for P0: 3

...

Output 2:

Deadlock cannot be avoided

RESULT:

Thus, the Banker's Algorithm has been implemented using a C program and tested for multiple cases.

EXP NO:16

REG.NO:

DATE:

NAME:

DYNAMIC MEMORY ALLOCATION

a) FIRST-FIT

AIM:

To implement the **First Fit Memory Allocation** algorithm, which allocates the first available memory block that is large enough to accommodate each process's memory requirements.

PROBLEM STATEMENT

Write a C program that simulates the **First Fit Memory Allocation** strategy. The program should assign memory blocks to a set of processes based on their memory requirements, selecting the first available block that is large enough. If a process cannot find a suitable memory block, it should be marked as "Not Allocated."

PROBLEM DESCRIPTION

The **First Fit Memory Allocation** algorithm assigns the first available memory block that can fit each process's size requirement.

- **Input:** Array of memory blocks and array of processes with memory sizes.
- **Output:** Allocation results, indicating the assigned block for each process or "Not Allocated" if no suitable block is available.
- **Process:** For each process, search for the first block large enough, allocate it, reduce the block's remaining size, and proceed to the next process.

CODE:

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

```
#include <string.h>
```

```
// Structure to represent a process
```

```
typedef struct {
```

```

    int id;        // Process ID

    int size;      // Size of the process
} Process;

// Function to allocate memory to blocks as per first fit algorithm
void firstFit(int blockSize[], int m, Process processes[], int n) {
    int allocation[n]; // Stores block id of the block allocated to a process
    memset(allocation, -1, sizeof(allocation)); // Initially, no block is assigned to any process

    // Pick each process and find the first suitable block according to its size
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processes[i].size) {
                allocation[i] = j; // Allocate block to process
                blockSize[j] -= processes[i].size; // Reduce available memory in this block
                break; // Move to the next process
            }
        }
    }
}

// Print the allocation results
printf("\nProcess ID\tProcess Size\tBlock No.\n");
for (int i = 0; i < n; i++) {
    printf(" %d\t\t%d\t\t", processes[i].id, processes[i].size);
    if (allocation[i] != -1) {
        printf("%d", allocation[i] + 1); // Print block number (1-indexed)
    } else {
        printf("Not Allocated");
    }
}
printf("\n");

```

```

    }
}

// Driver code
int main() {
    int blockSize[] = {100, 500, 200, 300, 600}; // Sizes of memory blocks
    Process processes[] = {{1, 212}, {2, 417}, {3, 112}, {4, 426}}; // Processes with sizes
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processes) / sizeof(processes[0]);

    firstFit(blockSize, m, processes, n);

    return 0;
}

```

SAMPLE INPUT AND OUTPUT:

```

int blockSize[] = {100, 500, 200, 300, 600}; // Sizes of memory blocks
Process processes[] = {{1, 212}, {2, 417}, {3, 112}, {4, 426}}; // Processes with sizes

```

Process ID	Process Size	Block No.
1	212	2
2	417	5
3	112	2
4	426	Not Allocated

PS D:\College\OS_LAB>

RESULT:

Thus, first-fit been programmed ,implemented and executed successfully.

b) BEST-FIT

AIM:

To implement a memory allocation system that uses the **Best Fit** algorithm to allocate requested memory blocks efficiently from a set of available memory blocks.

PROBLEM STATEMENT:

In memory management, allocation strategies aim to optimize the use of memory by fitting processes into memory blocks in the most efficient way possible. This program is designed to allocate memory dynamically using the Best Fit strategy, where a process is allocated to the smallest available memory block that can accommodate its size. Given multiple memory requests and a fixed set of memory blocks with varying sizes, the program should efficiently allocate memory to the requests, update the block states, and print the allocation results.

PROBLEM DESCRIPTION:

In this program, memory blocks of varying sizes are created and initially marked as free. Each block has properties indicating its size and whether it is allocated or free. When a request for memory allocation is made, the program follows these steps:

1. **Check Availability:** It scans all memory blocks to find free blocks that are large enough to fulfill the request.
2. **Best Fit Selection:** Among the free blocks that can accommodate the requested size, the program selects the block with the smallest size (Best Fit) to minimize unused memory.
3. **Allocation Process:**
 - a. If the selected block is larger than required, it allocates the requested memory portion from the block and adjusts the remaining size.
 - b. If the block size exactly matches the requested size, the entire block is allocated.
4. **Output:** The program provides feedback on whether the allocation was successful, the block chosen, and if any space remains after allocation.

If no suitable memory block is found for the requested size, the program outputs that the allocation has failed for that request.

CODE:

```
#include <stdio.h>

#define MAX_BLOCKS 100

typedef struct {
    int size; // Size of the memory block
    int isFree; // 1 if free, 0 if allocated
} MemoryBlock;

MemoryBlock blocks[MAX_BLOCKS]; // Array to hold memory blocks
int blockCount; // Number of memory blocks

// Function to allocate memory using the best fit algorithm
void bestFitAllocate(int requestSize) {
    int bestIndex = -1; // Index of the best block found
    int minSize = 9999; // A large number to compare

    printf("\nRequesting %d KB of memory...\n", requestSize);
    printf("Available Memory Blocks:\n");
    for (int i = 0; i < blockCount; i++) {
        printf("Block %d: Size = %d KB, Status = %s\n",
            i,
            blocks[i].size,
            blocks[i].isFree ? "Free" : "Allocated");
    }

    // Iterate through the blocks to find the best fit
    for (int i = 0; i < blockCount; i++) {
```



```

// Check if the block is free and large enough
if (blocks[i].isFree && blocks[i].size >= requestSize) {
    if (blocks[i].size < minSize) {
        minSize = blocks[i].size; // Update minSize
        bestIndex = i;           // Update bestIndex
    }
}

// Allocate if a suitable block was found
if (bestIndex != -1) {
    blocks[bestIndex].isFree = 0; // Mark as allocated
    printf("Best fit found at Block %d (Size = %d KB)\n", bestIndex,
blocks[bestIndex].size);

    // If the block is larger than requested, split it
    if (blocks[bestIndex].size > requestSize) {
        printf("Allocating %d KB from Block %d, remaining size will be %d KB\n",
            requestSize,
            bestIndex,
            blocks[bestIndex].size - requestSize);
        blocks[bestIndex].size -= requestSize; // Reduce size of the block
    } else {
        printf("Allocating entire Block %d (Size = %d KB)\n",
            bestIndex,
            blocks[bestIndex].size);
        blocks[bestIndex].size = 0; // Block is fully allocated
    }
} else {
    printf("No suitable block found for %d KB request\n", requestSize);
}

```

```
}  
  
// Example usage  
  
int main() {  
    // Initialize blocks  
  
    blockCount = 5;  
  
    blocks[0] = (MemoryBlock){100, 1}; // Block 1  
    blocks[1] = (MemoryBlock){500, 1}; // Block 2  
    blocks[2] = (MemoryBlock){200, 1}; // Block 3  
    blocks[3] = (MemoryBlock){300, 1}; // Block 4  
    blocks[4] = (MemoryBlock){600, 1}; // Block 5  
  
    // Request memory allocations  
  
    bestFitAllocate(212); // Allocate 212 KB  
  
    bestFitAllocate(100); // Allocate 100 KB  
  
    bestFitAllocate(50); // Allocate 50 KB  
  
    bestFitAllocate(400); // Attempt to allocate 400 KB (should fail)  
  
    return 0;  
}
```

SAMPLE INPUT AND OUTPUT:

```
Requesting 212 KB of memory...
Available Memory Blocks:
Block 0: Size = 100 KB, Status = Free
Block 1: Size = 500 KB, Status = Free
Block 2: Size = 200 KB, Status = Free
Block 3: Size = 300 KB, Status = Free
Block 4: Size = 600 KB, Status = Free
Best fit found at Block 3 (Size = 300 KB)
Allocating 212 KB from Block 3, remaining size will be 88 KB
```

```
Requesting 100 KB of memory...
Available Memory Blocks:
Block 0: Size = 100 KB, Status = Free
Block 1: Size = 500 KB, Status = Free
Block 2: Size = 200 KB, Status = Free
Block 3: Size = 88 KB, Status = Allocated
Block 4: Size = 600 KB, Status = Free
Best fit found at Block 0 (Size = 100 KB)
Allocating entire Block 0 (Size = 100 KB)
```

```
Requesting 50 KB of memory...
Available Memory Blocks:
Block 0: Size = 0 KB, Status = Allocated
Block 1: Size = 500 KB, Status = Free
Block 2: Size = 200 KB, Status = Free
Block 3: Size = 88 KB, Status = Allocated
Block 4: Size = 600 KB, Status = Free
Best fit found at Block 2 (Size = 200 KB)
Allocating 50 KB from Block 2, remaining size will be 150 KB
```

```
Requesting 400 KB of memory...
Available Memory Blocks:
```

```
Requesting 400 KB of memory...
Available Memory Blocks:
Block 0: Size = 0 KB, Status = Allocated
Block 1: Size = 500 KB, Status = Free
Block 2: Size = 150 KB, Status = Allocated
Block 3: Size = 88 KB, Status = Allocated
Block 4: Size = 600 KB, Status = Free
Best fit found at Block 1 (Size = 500 KB)
Allocating 400 KB from Block 1, remaining size will be 100 KB
```

RESULT:

Thus, best-fit been programmed, implemented and executed successfully.

c) **WORST-FIT**

AIM:

To implement a memory allocation system that uses the **Worst Fit** algorithm to allocate memory blocks efficiently by choosing the largest available memory block that can fulfill a given memory request.

PROBLEM STATEMENT:

Memory management in operating systems requires allocating processes into memory blocks while minimizing fragmentation and optimizing memory usage. In the **Worst Fit** strategy, the objective is to allocate a process to the largest available memory block that meets the requested size, with the aim of maximizing the number of larger blocks left available for future requests. This approach intends to leave smaller blocks unoccupied, which can potentially fulfill smaller requests more efficiently.

Given a set of fixed-size memory blocks, the program should dynamically allocate memory requests using the **Worst Fit** strategy and update the memory block states after each allocation attempt.

PROBLEM DESCRIPTION:

In this program, a set of memory blocks with varying sizes is initialized and marked as free. Each memory block has attributes indicating its size and status (free or allocated). When a memory request arrives, the program follows these steps:

1. **Scan for Eligible Blocks:** It checks all available memory blocks to find free blocks large enough to meet the requested memory size.
2. **Select Worst Fit Block:** Among the eligible blocks, the program selects the block with the largest size (Worst Fit) to maximize the chance of fitting future smaller requests into smaller memory blocks.
3. **Allocate Memory:**
 - a. If the selected block is larger than the requested size, the program allocates the requested portion and updates the block's remaining size.
 - b. If the selected block size exactly matches the request, the entire block is allocated to that request.

4. **Feedback:** The program provides output indicating whether the allocation was successful, which block was selected, and any remaining space in the block after allocation.

If no block is large enough to satisfy the request, the program outputs a failure message for that request.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX_BLOCKS 10 // Maximum number of memory blocks

// Structure to represent a memory block
struct MemoryBlock {
    int size;

    int originalSize; // To store the initial size of the block

    int isAllocated;
};

void worstFitAllocate(int processSize, struct MemoryBlock blocks[], int n) {
    int worstIndex = -1;

    // Find the index of the worst (largest) block that can fit the process
    for (int i = 0; i < n; i++) {
        if (!blocks[i].isAllocated && blocks[i].size >= processSize) {
            if (worstIndex == -1 || blocks[i].size > blocks[worstIndex].size) {
                worstIndex = i;
            }
        }
    }

    // If a suitable block was found, allocate it
    if (worstIndex != -1) {
        blocks[worstIndex].isAllocated = 1;

        printf("Process of size %d allocated to block %d with size %d\n",
            processSize, worstIndex + 1, blocks[worstIndex].originalSize);

        blocks[worstIndex].size -= processSize; // Reduce block size by allocated process size
    } else {
```

```

        printf("No suitable block found for process of size %d\n", processSize);
    }
}

int main() {
    int n, m;

    printf("Enter the number of memory blocks: ");
    scanf("%d", &n);
    struct MemoryBlock blocks[MAX_BLOCKS];

    // Input block sizes
    for (int i = 0; i < n; i++) {
        printf("Enter size of block %d: ", i + 1);
        scanf("%d", &blocks[i].size);
        blocks[i].originalSize = blocks[i].size; // Store the original size of the block
        blocks[i].isAllocated = 0; // Initially, no blocks are allocated
    }

    printf("Enter the number of processes: ");
    scanf("%d", &m);
    int processSizes[m];

    // Input process sizes
    for (int i = 0; i < m; i++) {
        printf("Enter size of process %d: ", i + 1);
        scanf("%d", &processSizes[i]);
    }

    // Allocate memory to processes using the Worst Fit algorithm
    for (int i = 0; i < m; i++) {

```

```

        worstFitAllocate(processSizes[i], blocks, n);
    }
    // Display remaining block sizes
    printf("\nRemaining block sizes:\n");
    for (int i = 0; i < n; i++) {
        printf("Block %d: %d\n", i + 1, blocks[i].size);
    }
    return 0;
}

```

SAMPLE INPUT AND OUTPUT:

```

Enter the number of memory blocks: 5
Enter size of block 1: 100
Enter size of block 2: 500
Enter size of block 3: 200
Enter size of block 4: 300
Enter size of block 5: 600
Enter the number of processes: 4
Enter size of process 1: 212
Enter size of process 2: 417
Enter size of process 3: 112
Enter size of process 4: 426
Process of size 212 allocated to block 5 with size 600
Process of size 417 allocated to block 2 with size 500
Process of size 112 allocated to block 4 with size 300
No suitable block found for process of size 426

Remaining block sizes:
Block 1: 100
Block 2: 83
Block 3: 200
Block 4: 188
Block 5: 388

```

RESULT:

Thus, worst-fit been programmed, implemented and executed successfully.

EXP NO: 17

REG NO:

DATE:

NAME:

PAGING HARDWARE IMPLEMENTATION

AIM:

To implement and simulate the paging hardware, including address translation, using C programming language.

OBJECTIVES:

1. Understand the paging mechanism and simulate the hardware mapping in memory management.
2. Implement a page table structure and simulate address translation.
3. Manage physical and virtual memory allocation using page tables.

PROBLEM DESCRIPTION:

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This experiment involves implementing a simple paging system that divides memory into pages, maintains a page table for logical-to-physical address translation, and demonstrates address translation by mapping virtual addresses to physical addresses. The program should allow the user to enter a virtual address, which will be translated to a physical address based on the page table.

THEORY:

Paging is a method of memory allocation that divides memory into small, fixed-size units called pages for virtual memory and frames for physical memory. Each process has a page table, which keeps track of the mapping from virtual pages to physical frames.

1. Divide the process's address space into pages
2. Use a page table to map these pages to frames in physical memory
3. Translate virtual addresses into physical addresses using the page table

SYSTEM CALLS USED:

1. malloc() – For memory allocation
2. free() – To deallocate memory

PROGRAM CODE:

```
#include <stdio.h>

#include <stdlib.h>

#define PAGE_SIZE 4096 // Define page size (4KB)
#define NUM_PAGES 4    // Define number of pages
#define NUM_FRAMES 4   // Define number of frames

int page_table[NUM_PAGES]; // Page table

// Function to simulate memory mapping
void initialize_page_table() {
    for (int i = 0; i < NUM_PAGES; i++) {
        page_table[i] = i; // Simple one-to-one mapping for simulation
    }
}

int translate_address(int virtual_address) {
    int page_number = virtual_address / PAGE_SIZE;
    int offset = virtual_address % PAGE_SIZE;

    if (page_number >= NUM_PAGES) {
        printf("Invalid virtual address: %d\n", virtual_address);
        return -1;
    }
}
```

```

    int frame_number = page_table[page_number];
    int physical_address = frame_number * PAGE_SIZE + offset;
    return physical_address;
}

int main() {
    initialize_page_table();
    int virtual_address;
    printf("Enter a virtual address to translate: ");
    scanf("%d", &virtual_address);
    int physical_address = translate_address(virtual_address);
    if (physical_address != -1) {
        printf("Virtual Address: %d, Physical Address: %d\n", virtual_address,
physical_address);
    }
    return 0;
}

```

SAMPLE INPUT AND OUTPUT:

Input

Enter a virtual address to translate: 8192

Output:

Virtual Address: 8192, Physical Address: 8192

Input:

Enter a virtual address to translate: 12288

Output:

Virtual Address: 12288, Physical Address: 12288

RESULT:

Therefore, the paging hardware was successfully simulated, implementing virtual address translation using a page table in C. This program demonstrates the conversion of a virtual address to a physical address, showing the workings of paging as a memory management technique

EXP NO: 18

REG NO:

DATE:

NAME:

PAGE REPLACEMENT ALGORITHMS

a) First In First Out [FIFO]

AIM:

To implement **FIFO Page Replacement** in C and analyze the number of page hits, page faults, and hit/fault ratios for a given reference string.

PROBLEM DESCRIPTION:

In **FIFO Page Replacement**, pages are loaded into memory frames in the order they are requested. If a new page needs to be added and memory is full, the **oldest page** (First-In-First-Out) is replaced. This program simulates the FIFO algorithm, tracks page hits and faults, and calculates performance metrics.

Library Functions Used:

1. **#include <stdlib.h>**: Provides functions for memory allocation (malloc()), deallocation (free()), and other utilities like sizeof().
2. **#include <unistd.h>**: Contains usleep() for adding delays to simulate real-time processes.

PROGRAM CODE:

```
#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h> // For malloc and free

void fifoPageReplacement(int pages[], int n, int capacity) {
    int *memory = (int *)malloc(capacity * sizeof(int)); // Dynamic memory allocation
    int front = 0; // Points to the oldest page (FIFO)
    int page_faults = 0; // Page fault counter
    int page_hits = 0; // Page hit counter
```

```

int size = 0;           // Current number of pages in memory

// Initialize memory to -1 (indicates empty slots)
for (int i = 0; i < capacity; i++) {
    memory[i] = -1;
}

printf("\n-----\n");
printf("| Page Request    | Memory State    | Status (Hit/Fault) |\n");
printf("-----\n");

// Process each page request
for (int i = 0; i < n; i++) {
    int page = pages[i]; // Current page request
    bool hit = false;

    // Check if the page is already in memory (page hit)
    for (int j = 0; j < capacity; j++) {
        if (memory[j] == page) {
            page_hits++; // Page hit detected
            hit = true;
            break;
        }
    }

    if (hit) {
        // If hit, just print the state of memory and move on
        printf("|    %3d    |", page);
        for (int j = 0; j < capacity; j++) {
            if (memory[j] != -1)
                printf("%3d ", memory[j]);
            else
                printf(" - ");
        }
    }
}

```

```

    }

    printf("      |   Hit      |\n");
} else {
    // If not hit, it's a page fault, so replace the oldest page (FIFO)

    page_faults++;

    if (size < capacity) {
        // If memory is not yet full, just add the page

        memory[size++] = page;
    } else {
        // Replace the oldest page (pointed by 'front')

        memory[front] = page;

        front = (front + 1) % capacity; // Update front pointer
    }

    // Print the state of memory after replacement

    printf("|      %3d      |", page);

    for (int j = 0; j < capacity; j++) {
        if (memory[j] != -1)
            printf("%3d ", memory[j]);
        else
            printf(" - ");
    }

    printf("      |   Fault      |\n");
}

}

printf("-----\n");

printf("\nSummary:\n");

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

printf("Total Page Hits   : %d\n", page_hits);

printf("Hit Ratio         : %.2f\n", (float)page_hits / (float)n);

printf("Fault Ratio        : %.2f\n", (float)page_faults / (float)n);

```

```

    free(memory); // Free dynamically allocated memory
}

int main() {
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2}; // Page reference string
    int n = sizeof(pages) / sizeof(pages[0]); // Number of page requests
    int capacity = 3; // Number of frames available in memory

    fifoPageReplacement(pages, n, capacity);

    return 0;
}

```

OUTPUT:

```

● zia@kez:~$ gcc fifoPR.c -o fifoPR
● zia@kez:~$ ./fifoPR

```

Page Request	Memory State	Status (Hit/Fault)
7	7 - -	Fault
0	7 0 -	Fault
1	7 0 1	Fault
2	2 0 1	Fault
0	2 0 1	Hit
3	2 3 1	Fault
0	2 3 0	Fault
4	4 3 0	Fault
2	4 2 0	Fault
3	4 2 3	Fault
0	0 2 3	Fault
3	0 2 3	Hit
2	0 2 3	Hit

```

Summary:
Total Page Faults : 10
Total Page Hits   : 3
Hit Ratio         : 0.23
Fault Ratio       : 0.77

```

RESULT:

Thus, Dining Philosophers Program has been implemented and run successfully.

b) LRU

AIM:

The aim of this program is to implement LRU (Least Recently Used) page replacement algorithm.

ALGORITHM:

☐ **Initialize memory frames:**

- Set each frame to -1 (indicating empty) and initialize variables to track page faults and reference times.

☐ **Input page references:**

- Accept the number of frames, number of pages, and the sequence of page references.

☐ **Process each page reference:**

- For each page:
 - **Check if the page is already in memory:**
 - If yes, update its last used time, and continue.
 - If no, proceed to step 4.

☐ **Handle page faults:**

- If there is an empty frame:
 - Insert the page into the empty frame, update the last used time, and increase the page fault count.
- If there is no empty frame:
 - Use the findLRU() function to identify the least recently used page.
 - Replace the identified page with the new page, update the time, and increment page faults.

☐ **Update memory status:**

- After each page reference, display the current status of frames (pages loaded in memory).

☐ **Calculate and display total page faults:**

- At the end, print the total count of page faults.

Program code:

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

```
int findLRU(int time[], int n) {  
    int i, minimum = time[0], pos = 0;  
    for (i = 1; i < n; ++i) {  
        if (time[i] < minimum) {  
            minimum = time[i];  
            pos = i;  
        }  
    }  
    return pos;  
}
```

```
int main() {  
    int n, frames, pages[30], time[10], flag1, flag2, i, j, pos, page_faults = 0;  
    int memory[10];  
  
    printf("Enter the number of frames: ");  
    scanf("%d", &frames);  
  
    printf("Enter the number of pages: ");  
    scanf("%d", &n);  
  
    printf("Enter the page references:\n");  
    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) {
```

```
    flag1 = flag2 = 0;
```

```
    for (j = 0; j < frames; ++j) {
```

```
        if (memory[j] == pages[i]) {
```

```
            flag1 = flag2 = 1;
```

```
            time[j] = i; // Update the last used time
```

```
            break;
```

```
        }
```

```
    }
```

```
if (flag1 == 0) {
```

```
    for (j = 0; j < frames; ++j) {
```

```
        if (memory[j] == -1) {
```

```
            page_faults++;
```

```
            memory[j] = pages[i];
```

```
            time[j] = i;
```

```
            flag2 = 1;
```

```
            break;
```

```
        }
```

```
    }
```

```
}
```

```
if (flag2 == 0) {
```

```
    pos = findLRU(time, frames);
```

```
    memory[pos] = pages[i];
```

```
    time[pos] = i;
```

```
    page_faults++;
```

```

    }

    printf("\nMemory status after referencing page %d: ", pages[i]);
    for (j = 0; j < frames; ++j) {
        if (memory[j] != -1)
            printf("%d ", memory[j]);
        else
            printf("- ");
    }
}

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

```

TEST CASE:

```

kathir@LAPTOP-3GI0DMDQ:~$ ./a.out
Enter the number of frames: 5
Enter the number of pages: 7
Enter the page references:
1
2
3
4
5
6
7

Memory status after referencing page 1: 1 - - - -
Memory status after referencing page 2: 1 2 - - -
Memory status after referencing page 3: 1 2 3 - -
Memory status after referencing page 4: 1 2 3 4 -
Memory status after referencing page 5: 1 2 3 4 5
Memory status after referencing page 6: 6 2 3 4 5
Memory status after referencing page 7: 6 7 3 4 5

Total Page Faults = 7

```

RESULT:

Thus, the LRU algorithm has been implemented using C program.

EXP NO: 19

REG NO:

DATE:

NAME:

FILE ALLOCATION TECHNIQUES

a) Single-level

PROBLEM STATEMENT:

Write a C program that lists the contents of a specified directory in a single-level tree format. The program should print all files and folders within the specified directory, without delving into any subdirectories.

PROBLEM DESCRIPTION:

The program should:

1. Accept a directory path as a command-line argument.
2. Open the specified directory.
3. List all entries (files and directories) contained within that directory at the top level.
4. Skip the current (.) and parent (..) directories.
5. Print the name of each entry prefixed by a |-- symbol to indicate a tree-like structure.
6. If no directory path is provided as an argument, the program should default to listing the contents of the current directory.

ALGORITHM:

1. Input Handling:

- Check if a directory path is provided as a command-line argument.
- If a path is provided, use it as the target directory; otherwise, default to the current directory (".").

2. Directory Opening:

- Attempt to open the specified directory using the opendir function.
- If the directory cannot be opened (e.g., due to insufficient permissions or invalid path), print an error message and exit.

3. Reading Directory Entries:

- For each entry in the directory, read it using readdir.
- Skip entries that represent the current directory (".") or parent directory ("..").

4. **Print Entry Names:**

- For each valid entry, print its name prefixed with |--, creating a simple visual structure.

5. **Close Directory:**

- After listing all entries, close the directory to free up resources.

6. **Output:**

- Display the contents of the specified directory in a single-level format.

PROGRAM:

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

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

```
#include <dirent.h>
```

```
#include <sys/stat.h>
```

```
#include <string.h>
```

```
void list_directory(const char *path) {
```

```
    struct dirent *entry;
```

```
    DIR *dir = opendir(path);
```

```
    // Return if directory cannot be opened
```

```
    if (!dir) {
```

```
        perror("Unable to open directory");
```

```
        return;
```

```
    }
```

```
    printf("Contents of directory: %s\n", path);
```

```
    // Iterate through each entry in the directory
```

```
    while ((entry = readdir(dir)) != NULL) {
```

```
        // Skip "." and ".." directories
```

```
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
```

```
            continue;
```

```

    }

    // Print entry name
    printf("|-- %s\n", entry->d_name);
}

closedir(dir);
}

int main(int argc, char *argv[]) {
    const char *path;

    // Check if directory path is provided as an argument
    if (argc < 2) {
        path = "."; // Default to current directory
    } else {
        path = argv[1];
    }

    list_directory(path);

    return 0;
}

```

Output:

```
Contents of directory: .
|-- producerconumer.c
|-- prio.c
|-- srtf.c
|-- bankers.c
|-- sjf.c
|-- tree
|-- tree.c
|-- premprio
|-- preemptiveprio
|-- oddeven.sh
|-- prio
|-- two_level
|-- srtf
|-- sjf
|-- palindrome.sh
|-- reverse.sh
|-- roundrobin.c
|-- vowels.sh
|-- bankers
|-- newfile1.txt
|-- fcfs.c
|-- a.out
|-- dining_p_s.c
|-- prio.c.txt
|-- armstrong.sh
|-- two_level.c
|-- dining_p_s
```

RESULT:

Thus, the program was executed successfully

b. Two – level File organization Technique

AIM:

To implement a two-level file organization technique that organizes files within user directories. Each user has their own directory, which contains a list of files. This two-level structure helps to organize and manage files in a structured manner.

ALGORITHM:

Step 1: Define structures.

- Create a File structure with attributes name and size.
- Create a UserDirectory structure with attributes userName, an array of Files, and fileCount.

Step 2: Initialize the root directory.

- Declare an array rootDir of type UserDirectory to hold all user directories.
- Initialize userCount to 0 to track the number of created user directories.

Step 3: Create a new user directory (createUserDirectory function).

- Check if userCount is less than the maximum allowed users (MAX_USERS).
 - If yes, copy userName into rootDir[userCount].
 - Set fileCount to 0 for this user.
 - Increment userCount.
 - Print a success message.
- If the maximum user limit is reached, print an error message.

Step 4: Create a file in a user directory (createFile function).

- Loop through rootDir to find the directory that matches userName.
- If the directory is found:
 - Check if fileCount is less than the maximum allowed files (MAX_FILES).
 - If yes, add a new file with fileName and size in the files array of rootDir[i].
 - Increment fileCount.
 - Print a success message.
 - If file limit is reached, print an error message.
- If the user directory is not found, print an error message.

Step 5: Display files in a specific user directory (displayFiles function).

- Loop through rootDir to find the directory that matches userName.
- If the directory is found, print the names and sizes of all files in that directory.
- If the directory is not found, print an error message.

Step 6: Display all user directories and their files (displayAll function).

- Print a header "Root Directory".
- Loop through each user directory in rootDir.
 - For each user directory, print userName.
 - Loop through each file in the directory and print the file's name and size.

Step 7: Demonstrate functionality in main function.

- Call createUserDirectory to create directories for "user1" and "user2".
- Call createFile to add files to these user directories.
- Use displayFiles to show files in a specific user directory.
- Use displayAll to list all user directories and their files.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_USERS 5

#define MAX_FILES 5

#define MAX_NAME_LEN 20


// Structure for a file
typedef struct {
    char name[MAX_NAME_LEN];
    int size; // Size of the file in bytes
} File;
```



```

// Structure for a user directory
typedef struct {
    char userName[MAX_NAME_LEN];
    File files[MAX_FILES];
    int fileCount;
} UserDirectory;

// Root directory containing user directories
UserDirectory rootDir[MAX_USERS];
int userCount = 0;

// Function to create a new user directory
void createUserDirectory(char *userName) {
    if (userCount < MAX_USERS) {
        strcpy(rootDir[userCount].userName, userName);
        rootDir[userCount].fileCount = 0;
        userCount++;
        printf("User directory '%s' created.\n", userName);
    } else {
        printf("Cannot create user directory. Max limit reached.\n");
    }
}

// Function to create a file in a user's directory
void createFile(char *userName, char *fileName, int size) {
    for (int i = 0; i < userCount; i++) {
        if (strcmp(rootDir[i].userName, userName) == 0) {
            if (rootDir[i].fileCount < MAX_FILES) {
                File *newFile = &rootDir[i].files[rootDir[i].fileCount++];
                strcpy(newFile->name, fileName);
            }
        }
    }
}

```

```

        newFile->size = size;

        printf("File '%s' created in user directory '%s'.\n", fileName, userName);

        return;
    } else {

        printf("Cannot create file. Max files limit reached in '%s' directory.\n", userName);

        return;

    }

}

}

printf("User directory '%s' not found.\n", userName);
}

```

// Function to display all files in a user's directory

```

void displayFiles(char *userName) {
    for (int i = 0; i < userCount; i++) {
        if (strcmp(rootDir[i].userName, userName) == 0) {
            printf("Files in user directory '%s':\n", userName);

            for (int j = 0; j < rootDir[i].fileCount; j++) {
                printf("  %s (%d bytes)\n", rootDir[i].files[j].name, rootDir[i].files[j].size);
            }

            return;
        }
    }

    printf("User directory '%s' not found.\n", userName);
}

```

// Function to display all user directories and files

```

void displayAll() {
    printf("Root Directory:\n");

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

```

```

        printf("User Directory: %s\n", rootDir[i].userName);
        for (int j = 0; j < rootDir[i].fileCount; j++) {
            printf("  File: %s (%d bytes)\n", rootDir[i].files[j].name, rootDir[i].files[j].size);
        }
    }
}

```

```

int main() {
    createUserDirectory("user1");
    createUserDirectory("user2");

    createFile("user1", "file1.txt", 100);
    createFile("user1", "file2.txt", 200);
    createFile("user2", "fileA.txt", 150);

    printf("\nDisplaying files in user1's directory:\n");
    displayFiles("user1");

    printf("\nDisplaying all directories and files:\n");
    displayAll();

    return 0;
}

```

OUTPUT:

```
User directory 'user1' created.
User directory 'user2' created.
File 'file1.txt' created in user directory 'user1'.
File 'file2.txt' created in user directory 'user1'.
File 'fileA.txt' created in user directory 'user2'.

Displaying files in user1's directory:
Files in user directory 'user1':
  file1.txt (100 bytes)
  file2.txt (200 bytes)

Displaying all directories and files:
Root Directory:
User Directory: user1
  File: file1.txt (100 bytes)
  File: file2.txt (200 bytes)
User Directory: user2
  File: fileA.txt (150 bytes)
```

RESULT:

This output demonstrates that the two-level file organization technique is effectively managing files within user-specific directories.

c. Tree-Structured

PROBLEM STATEMENT:

Write a C program that displays the contents of a specified directory in a multi-level tree format, similar to the tree command in Unix-based systems. The program should recursively list all files and subdirectories within the specified directory, visually representing the directory hierarchy.

PROBLEM DESCRIPTION:

The program should:

1. Accept a directory path as a command-line argument.
2. Open the specified directory and list all entries (files and directories) within it.
3. For each entry, check if it is a file or a directory.
4. If it is a file, print the filename.
5. If it is a directory, print the directory name and recursively explore its contents.
6. Visually structure the output to indicate the depth level of each file or directory.
7. Skip special entries representing the current directory (".") and the parent directory ("..").
8. If no directory path is provided, the program should default to the current directory.

PROBLEM ALGORITHM:

1. **Input Handling:**
 - Check if a directory path is provided as a command-line argument.
 - If a path is provided, use it as the target directory; otherwise, default to the current directory (".").
2. **Directory Opening:**
 - Attempt to open the specified directory using opendir.
 - If the directory cannot be opened, return and do not proceed further.
3. **Reading Directory Entries:**
 - For each entry in the directory, read it using readdir.
 - Skip entries that represent the current directory (".") or parent directory ("..").
4. **Print Entry Names with Indentation:**
 - Print each entry name prefixed with |-- to create a tree-like structure.
 - Use an increasing indentation level based on the depth level to visually indicate hierarchy.

5. Recursive Directory Traversal:

- For each directory entry, construct its full path.
- If the entry is a directory, call the function recursively with the increased depth level to explore and print its contents.

6. Close Directory:

- After listing all entries, close the directory to free up resources.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <sys/stat.h>

void print_directory_tree(const char *base_path, int depth) {
    struct dirent *entry;
    DIR *dir = opendir(base_path);

    // Return if directory cannot be opened
    if (!dir) {
        return;
    }

    while ((entry = readdir(dir)) != NULL) {
        // Skip "." and ".." directories
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }

        // Print indentation based on depth level
        for (int i = 0; i < depth; i++) {
            printf(" ");
```

```

    }

    printf("|-- %s\n", entry->d_name);

    // Check if entry is a directory
    char path[1024];
    struct stat statbuf;
    snprintf(path, sizeof(path), "%s/%s", base_path, entry->d_name);
    if (stat(path, &statbuf) == 0 && S_ISDIR(statbuf.st_mode)) {
        // Recursive call with increased depth
        print_directory_tree(path, depth + 1);
    }
}

closedir(dir);
}

int main(int argc, char *argv[]) {
    // Check if directory path is provided as an argument
    const char *path;
    if (argc < 2) {
        path = "."; // Default to current directory
    } else {
        path = argv[1];
    }

    printf("%s\n", path);
    print_directory_tree(path, 0);

    return 0;
}

```

INPUT/OUTPUT:

```
.  
|-- ex1  
  |-- bankers.c  
  |-- oddeven.sh  
  |-- palindrome.sh  
  |-- bankers  
  |-- newfile1.txt  
  |-- fcfs.c  
  |-- a.out  
  |-- dining_p_s.c  
  |-- armstrong.sh  
  |-- dining_p_s  
  |-- newfile.txt  
  |-- commands  
  |-- fcfs  
  |-- fibonacci.sh  
  |-- commands.c  
  |-- fileread  
  |-- fileread.c  
  |-- factorical.sh  
|-- tree  
|-- tree.c  
|-- single_level  
|-- ex3  
  |-- srtf.c  
  |-- sjf.c  
  |-- tree  
  |-- two_level  
  |-- srtf  
  |-- sjf  
  |-- roundrobin.c  
  |-- vowels.sh  
  |-- two_level.c  
  |-- single_level  
  |-- roundrobin  
  |-- sum.sh
```

RESULT:

Thus, the program has been executed successfully

d. Acyclic File organization Technique

AIM:

The aim of this program is to implement an acyclic file organization technique, where files and directories are organized in a hierarchical structure without any cyclic references. This structure supports the concept of directory shortcuts (links), allowing nodes to be shared across different paths while maintaining an acyclic nature.

ALGORITHM:

Step 1: Define Node structure.

- Create an enum named NodeType with two values: FILE_TYPE for files and DIRECTORY_TYPE for directories.
- Define a Node structure with the following attributes:
 - name (character array) to store the name of the file or directory.
 - type (NodeType) to distinguish between files and directories.
 - children (array of pointers to Node) to store the children of a directory.
 - childCount (integer) to count the number of children.

Step 2: Initialize the root directory.

- Declare a global pointer root of type Node* to represent the root directory node.

Step 3: Create a new node (function createNode).

- Allocate memory for a new Node.
- Set name and type of the new node according to the input parameters.
- Set childCount to 0 to start with no children.
- Return the pointer to this newly created node.

Step 4: Add a child node to a directory (function addChild).

- Check if the parent node's type is DIRECTORY_TYPE.
 - If it is not, print an error message and exit the function.
- If parent->childCount is less than MAX_CHILDREN, proceed to add the child:
 - Add child to the children array of parent.
 - Increment parent->childCount.
 - Print a success message indicating the addition of the child to the parent.
- If the MAX_CHILDREN limit is reached for the parent, print an error message.

Step 5: Create a link (shortcut) to an existing node (function `createLink`).

- Use the `addChild` function to add an existing node as a child to a directory node, thereby creating an acyclic link (shortcut).

Step 6: Print the directory structure recursively (function `printStructure`).

- Print the name of the current node, with appropriate indentation based on level.
- If the node is a directory, loop through each child:
 - Recursively call `printStructure` on each child, increasing the indentation level to represent the hierarchy visually.

Step 7: Main function.

- Initialize the root directory node by calling `createNode` with "root" as the directory name and `DIRECTORY_TYPE`.
- Create additional directories and files using `createNode`.
- Use `addChild` to build the hierarchical structure, adding directories and files as needed.
- Use `createLink` to create a link to an existing directory (e.g., link docs to home).
- Call `printStructure` to display the entire file system structure in a hierarchical, acyclic format.

CODE:

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

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

```
#include <string.h>
```

```
#define MAX_NAME_LEN 50
```

```
#define MAX_CHILDREN 10
```

```
typedef enum { FILE_TYPE, DIRECTORY_TYPE } NodeType;
```

```
typedef struct Node {
```

```
    char name[MAX_NAME_LEN];
```

```
    NodeType type;
```

```
    struct Node *children[MAX_CHILDREN];
```

```

    int childCount;
} Node;

// Root directory
Node *root;

// Function to create a new node (file or directory)
Node* createNode(const char *name, NodeType type) {
    Node *newNode = (Node*)malloc(sizeof(Node));
    strcpy(newNode->name, name);
    newNode->type = type;
    newNode->childCount = 0;
    return newNode;
}

// Function to add a child to a directory node
void addChild(Node *parent, Node *child) {
    if (parent->type != DIRECTORY_TYPE) {
        printf("Error: Cannot add a child to a file.\n");
        return;
    }
    if (parent->childCount < MAX_CHILDREN) {
        parent->children[parent->childCount++] = child;
        printf("Added '%s' to directory '%s'.\n", child->name, parent->name);
    } else {
        printf("Error: Maximum children limit reached for '%s'.\n", parent->name);
    }
}

// Function to create a link to an existing node (like a shortcut)

```

```
void createLink(Node *parent, Node *existingNode) {  
    addChild(parent, existingNode);  
}
```

```
// Function to print the directory structure
```

```
void printStructure(Node *node, int level) {  
    for (int i = 0; i < level; i++) printf(" ");  
    printf("|-- %s\n", node->name);  
    if (node->type == DIRECTORY_TYPE) {  
        for (int i = 0; i < node->childCount; i++) {  
            printStructure(node->children[i], level + 1);  
        }  
    }  
}
```

```
int main() {
```

```
    // Initialize root directory
```

```
    root = createNode("root", DIRECTORY_TYPE);
```

```
    // Create directories and files
```

```
    Node *home = createNode("home", DIRECTORY_TYPE);
```

```
    Node *user = createNode("user", DIRECTORY_TYPE);
```

```
    Node *docs = createNode("docs", DIRECTORY_TYPE);
```

```
    Node *file1 = createNode("file1.txt", FILE_TYPE);
```

```
    Node *file2 = createNode("file2.txt", FILE_TYPE);
```

```
    // Build the structure
```

```
    addChild(root, home);
```

```
    addChild(home, user);
```

```
    addChild(user, docs);
```

```
addChild(docs, file1);
addChild(docs, file2);

// Create a link to 'docs' from 'home' (creates acyclic graph)
createLink(home, docs);

// Print the file system structure
printf("File System Structure:\n");
printStructure(root, 0);

return 0;
}
```

OUTPUT:

```
Added 'home' to directory 'root'.
Added 'user' to directory 'home'.
Added 'docs' to directory 'user'.
Added 'file1.txt' to directory 'docs'.
Added 'file2.txt' to directory 'docs'.
Added 'docs' to directory 'home'.
File System Structure:
|-- root
    |-- home
        |-- user
            |-- docs
                |-- file1.txt
                |-- file2.txt
        |-- docs
            |-- file1.txt
            |-- file2.txt
```

RESULT:

This organization allows the file system to manage hierarchical relationships with shortcuts, facilitating efficient file access and organization in an acyclic structure.

EXP NO: 20

REG NO:

DATE:

NAME:

FILE ALLOCATION STRATEGIES

a) Contiguous Allocation

AIM:

To write a C program to perform file allocation on a disk using contiguous memory allocation.

ALGORITHM:

- a) **Initialize Disk:** Sets up the disk with a specified number of blocks, marking all blocks as free initially.
- b) **Allocate File:** Searches for a contiguous sequence of free blocks to store a file of a given size; if found, marks those blocks as used.
- c) **Free File:** Releases the blocks occupied by a file, making them available for future allocations.
- d) **Display Disk Status:** Outputs the current status of each block, showing whether it is free or used.
- e) **Main Execution:** Initializes the disk, performs file allocation and deallocation, and displays disk status after each operation to track changes.

PROGRAM CODE:

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

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

```
#include <string.h>
```

```
struct file {  
    char name[50];  
    int size;
```

```
    int start_block;  
};
```

```
struct disk {  
    int total_blocks;  
    int *used_blocks;  
    int *free_blocks;  
};
```

```
void init_disk(struct disk *d, int total_blocks) {  
    d->total_blocks = total_blocks;  
    d->used_blocks = (int *)calloc(total_blocks, sizeof(int));  
    d->free_blocks = (int *)calloc(total_blocks, sizeof(int));  
    for (int i = 0; i < total_blocks; i++) {  
        d->free_blocks[i] = 1;  
    }  
}
```

```
struct file *allocate_file(struct disk *d, char *name, int size) {  
    int start_block = -1;  
    int contiguous_blocks = 0;  
  
    for (int i = 0; i < d->total_blocks; i++) {  
        if (d->free_blocks[i] == 1) {  
            if (start_block == -1) {  
                start_block = i;  
            }  
            contiguous_blocks++;  
  
            if (contiguous_blocks == size) {
```

```

        break;
    }
} else {
    start_block = -1;
    contiguous_blocks = 0;
}
}

if (contiguous_blocks != size) {
    return NULL;
}

for (int i = start_block; i < start_block + size; i++) {
    d->used_blocks[i] = 1;
    d->free_blocks[i] = 0;
}

struct file *f = (struct file *)malloc(sizeof(struct file));
strcpy(f->name, name);
f->size = size;
f->start_block = start_block;

return f;
}

void free_file(struct disk *d, struct file *f) {
    for (int i = f->start_block; i < f->start_block + f->size; i++) {
        d->used_blocks[i] = 0;
        d->free_blocks[i] = 1;
    }
}

```



```

    free(f);
}

void print_disk_status(struct disk *d) {
    printf("\nDisk Status:\n");
    printf("Block\tStatus\n");
    for (int i = 0; i < d->total_blocks; i++) {
        printf("%d\t%s\n", i, d->used_blocks[i] ? "Used" : "Free");
    }
}

```

```

int main() {
    struct disk d;
    init_disk(&d, 15);
    struct file *f1 = allocate_file(&d, "file1", 3);
    if (f1 == NULL) {
        printf("Could not allocate space for file1\n");
        exit(1);
    }
    print_disk_status(&d);

    struct file *f2 = allocate_file(&d, "file2", 5);
    if (f2 == NULL) {
        printf("Could not allocate space for file2\n");
        exit(1);
    }
    print_disk_status(&d);

    free_file(&d, f1);
}

```

```
print_disk_status(&d);
```

```
struct file *f3 = allocate_file(&d, "file3", 4);
```

```
if (f3 == NULL) {
```

```
    printf("Could not allocate space for file3\n");
```

```
    exit(1);
```

```
}
```

```
print_disk_status(&d);
```

```
free_file(&d, f2);
```

```
print_disk_status(&d);
```

```
free_file(&d, f3);
```

```
print_disk_status(&d);
```

```
free(d.used_blocks);
```

```
free(d.free_blocks);
```

```
return 0;
```

```
}
```

TEST CASE:

Disk Status:

Available Space: 100%

Block	Status
0	Used
1	Used
2	Used
3	Free
4	Free
5	Free
6	Free
7	Free
8	Free
9	Free
10	Free
11	Free
12	Free
13	Free
14	Free

Disk Status:

Available Space: 100%

Block	Status
0	Used
1	Used
2	Used
3	Used
4	Used
5	Used
6	Used
7	Used
8	Free
9	Free
10	Free
11	Free
12	Free
13	Free
14	Free

Disk Status:

Available Space: 100%

Block	Status
0	Free
1	Free

File Actions Edit View Help

2

Free

3

Used

4

Used

5

Used

6

Used

7

Used

8

Free

9

Free

10

Free

11

Free

12

Free

13

Free

14

Free

Disk Status:

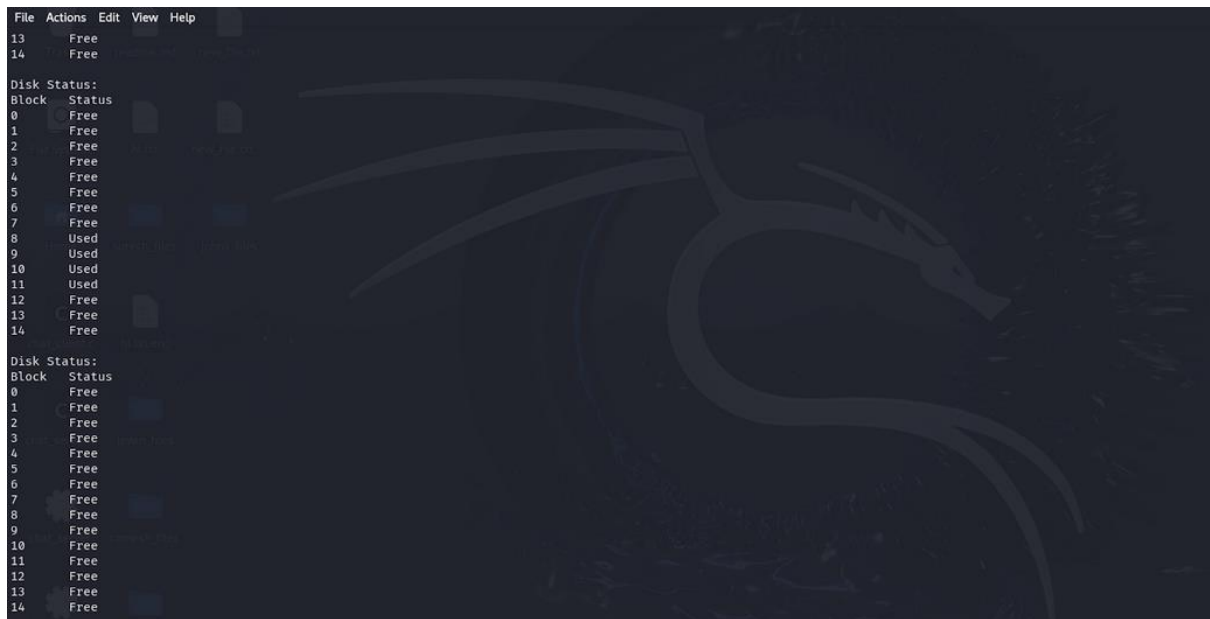
Available Space: 100%

Block	Status
0	Free
1	Free
2	Free
3	Used
4	Used
5	Used
6	Used
7	Used
8	Used
9	Used
10	Used
11	Used
12	Free
13	Free
14	Free

Disk Status:

Available Space: 100%

Block	Status
0	Free
1	Free
2	Free
3	Free
4	Free
5	Free
6	Free



RESULT:

Thus, a program using contiguous memory allocation has been executed successfully.

b) Linked Allocation

AIM:

To write a C program to perform file allocation on a disk using linked memory allocation.

ALGORITHM:

- a) **Initialize Blocks:** Set up an array of 50 blocks, marking all as free
- b) Create a queue to hold all pages in memory
- c) When the page is required replace the page at the head of the queue
- d) Now the new page is inserted at the tail of the queue
- e) Create a stack
- f) When the page fault occurs replace page present at the bottom of the stack
- g) Stop the allocation.

PROGRAM CODE:

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

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

```
int main() {
```

```
    int f[50], p, i, st, len, j, c, k, a;
```

```
    // Initialize all blocks as free (0)
```

```
for(i = 0; i < 50; i++)
```

```
    f[i] = 0;
```

```
printf("Enter how many blocks already allocated: ");
```

```
scanf("%d", &p);
```

```
printf("Enter blocks already allocated: ");
```

```
for(i = 0; i < p; i++) {
```

```
    scanf("%d", &a);
```

```
    f[a] = 1;
```

```
}
```

```
while (1) {
```

```
    printf("Enter index starting block and length: ");
```

```
    scanf("%d%d", &st, &len);
```

```
    k = len;
```

```
    if (f[st] == 0) {
```

```
        for (j = st; j < (st + k); j++) {
```

```
            if (f[j] == 0) {
```

```
                f[j] = 1;
```

```
                printf("%d----->%d\n", j, f[j]);
```

```
            } else {
```

```
                printf("%d Block is already allocated \n", j);
```

```
                k++;
```

```
            }
```

```
        }
```

```
    } else {
```

```
        printf("%d starting block is already allocated \n", st);
```

```
    }
```

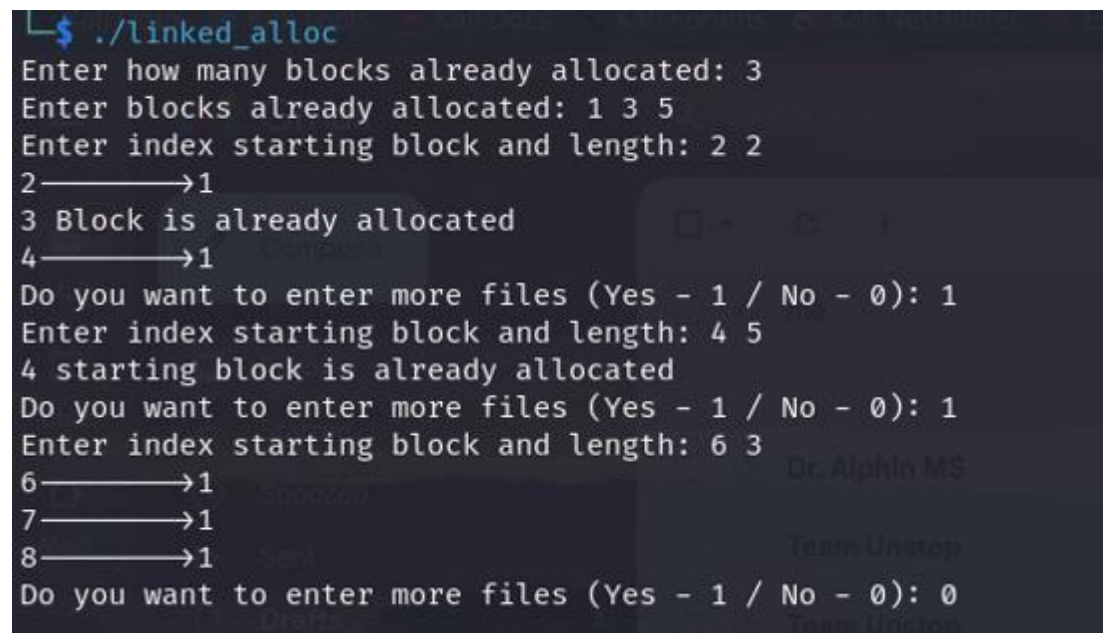
```

printf("Do you want to enter more files (Yes - 1 / No - 0): ");
scanf("%d", &c);
if (c == 0)
    break;
}

return 0;
}

```

TEST CASE:



```

$ ./linked_alloc
Enter how many blocks already allocated: 3
Enter blocks already allocated: 1 3 5
Enter index starting block and length: 2 2
2————→1
3 Block is already allocated
4————→1
Do you want to enter more files (Yes - 1 / No - 0): 1
Enter index starting block and length: 4 5
4 starting block is already allocated
Do you want to enter more files (Yes - 1 / No - 0): 1
Enter index starting block and length: 6 3
6————→1
7————→1
8————→1
Do you want to enter more files (Yes - 1 / No - 0): 0

```

RESULT:

Thus, a program using linked memory allocation has been executed successfully.

c) Indexed Allocation

AIM:

To write a C program to perform file allocation on a disk using indexed memory allocation.

ALGORITHM:

- a) **Initialize Blocks:** Set up an array of 50 blocks, marking each as free
- b) **Input Index Block:** Ask the user to enter an index block (ind).
 - If ind is already allocated, prompt the user to enter a different index.
- c) **Enter Blocks for File:**
 - Ask the user for the number of blocks required (n) for the file at index ind.
 - Input the specific block numbers, checking if each block is free.
- d) **Allocate Blocks:**
 - If all requested blocks are free, mark them as allocated.
 - Print a summary showing the file index and its allocated blocks.
 - If any block is already allocated, ask the user to re-enter blocks for this file.
- e) **Repeat or Exit:**
 - Ask if the user wants to allocate more files. If yes, return to step 2; if no, exit the program.

PROGRAM CODE:

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

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



```

int main() {
    int f[50], index[50], i, n, st, len, j, c, k, ind, count = 0;

    for (i = 0; i < 50; i++)
        f[i] = 0;

x:
    printf("Enter the index block: ");
    scanf("%d", &ind);

    if (f[ind] != 1) {
        printf("Enter number of blocks needed and number of files for the index %d on the
disk:\n", ind);
        scanf("%d", &n);
    } else {
        printf("%d index is already allocated\n", ind);
        goto x;
    }

y:
    count = 0;
    for (i = 0; i < n; i++) {
        scanf("%d", &index[i]);
        if (f[index[i]] == 0)
            count++;
    }
    if (count == n) {
        for (j = 0; j < n; j++)
            f[index[j]] = 1;

        printf("Allocated\n");
    }
}

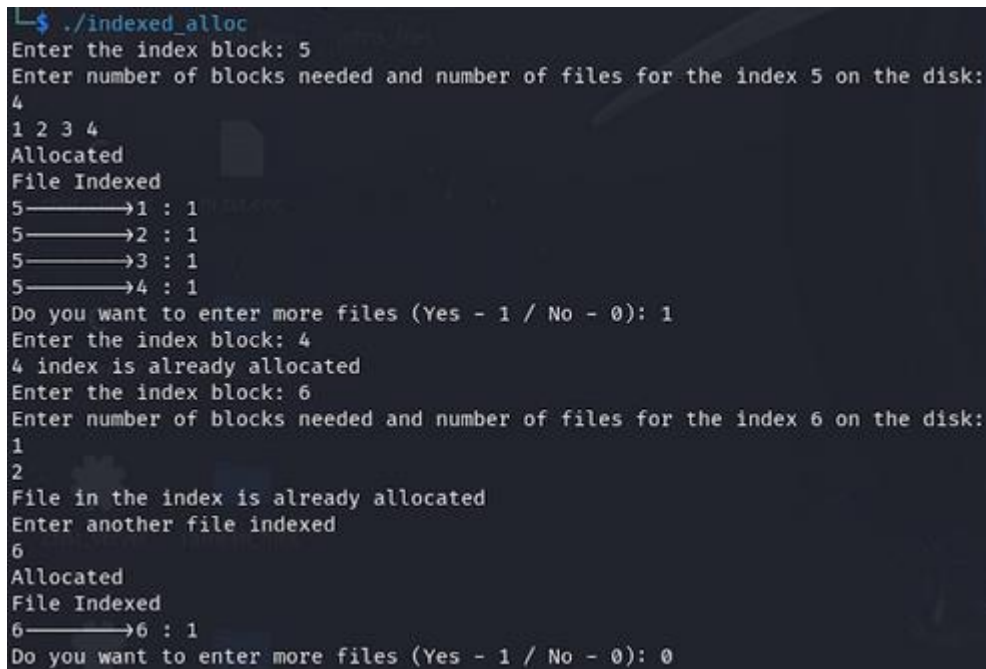
```

```

printf("File Indexed\n");
for (k = 0; k < n; k++)
    printf("%d----->%d : %d\n", ind, index[k], f[index[k]]);
} else {
    printf("File in the index is already allocated\n");
    printf("Enter another file indexed\n");
    goto y;
}
printf("Do you want to enter more files (Yes - 1 / No - 0): ");
scanf("%d", &c);
if (c == 1)
    goto x;
else
    exit(0);
return 0;
}

```

TEST CASE:



```

$ ./indexed_alloc
Enter the index block: 5
Enter number of blocks needed and number of files for the index 5 on the disk:
4
1 2 3 4
Allocated
File Indexed
5----->1 : 1
5----->2 : 1
5----->3 : 1
5----->4 : 1
Do you want to enter more files (Yes - 1 / No - 0): 1
Enter the index block: 4
4 index is already allocated
Enter the index block: 6
Enter number of blocks needed and number of files for the index 6 on the disk:
1
2
File in the index is already allocated
Enter another file indexed
6
Allocated
File Indexed
6----->6 : 1
Do you want to enter more files (Yes - 1 / No - 0): 0

```

RESULT:

Thus, a program using indexed memory allocation has been executed successfully.

