Os programs and outputs

1) a) Priority Scheduling algorithm

program :

```
#include<stdio.h>

struct Process {

        int id;

        int burst_time;

        int priority;

        int waiting_time;

        int turnaround_time;

};

void priorityScheduling(struct Process processes[], int n) {

        int total_waiting_time = 0, total_turnaround_time = 0;

        float average_waiting_time, average_turnaround_time;

        // Calculate waiting time for each process

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

                processes[i].waiting_time = total_waiting_time;

                total_waiting_time += processes[i].burst_time;

        }

        // Calculate turnaround time for each process

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

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

                total_turnaround_time += processes[i].turnaround_time;

        }

        // Calculate averages
```

```
    average_waiting_time = (float)total_waiting_time / n;
    average_turnaround_time = (float)total_turnaround_time / n;
 / Display results
    printf("Process ID\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time,
               processes[i].priority, processes[i].waiting_time, processes[i].turnaround_time);
    }
    printf("\nAverage Waiting Time: %.2f\n", average_waiting_time);
    printf("Average Turnaround Time: %.2f\n", average_turnaround_time);
}
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];
    printf("Enter process details:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        processes[i].id = i + 1;
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        printf("Priority: ");
        scanf("%d", &processes[i].priority);
    }
```

```
    priorityScheduling(processes, n);

    return 0;

}
```

output:

```
Enter the number of processes: 5
Enter process details:
Process 1:
Burst Time: 2
Priority: 1
Process 2:
Burst Time: 5
Priority: 2
Process 3:
Burst Time: 4
Priority: 3
Process 4:
Burst Time: 6
Priority: 4
Process 5:
Burst Time: 7
Priority: 5
Process ID      Burst Time      Priority        Waiting Time    Turnaround Time
1               2               1               0               2
2               5               2               2               7
3               4               3               7               11
4               6               4               11              17
5               7               5               17              24

Average Waiting Time: 4.80
Average Turnaround Time: 12.20
```

b) find if a given number is odd or even:

program:

```
#!/bin/bash

echo "Enter a number:"

read num

if [ $((num % 2)) -eq 0 ]; then

    echo "$num is even."

else

    echo "$num is odd."

fi
```

output:

```
Enter a number:
5
5 is odd.
```

2)a) First Fit Memory Allocation Method :

program:

```
#include <stdio.h>

#define MAX_BLOCKS 50

#define MAX_JOBS 50

void firstFit(int blocks[], int m, int jobs[], int n) {

    int allocation[MAX_JOBS];

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

        allocation[i] = -1;

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

            if (blocks[j] >= jobs[i]) {

                allocation[i] = j;

                blocks[j] -= jobs[i];

               break;}}}

    printf("Job Number\tJob Size\tBlock Number\n");

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

        printf("%d\t\t%d\t\t", i + 1, jobs[i]);

        if (allocation[i] != -1) {

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

        } else {

            printf("Not Allocated\n");              }

   }

}

int main() {

    int m, n;
```

```
    printf("Enter the number of memory blocks: ");
    scanf("%d", &m);
    int blocks[MAX_BLOCKS];
    printf("Enter the sizes of memory blocks:\n");
    for (int i = 0; i < m; i++) {
        scanf("%d", &blocks[i]);
    }
    printf("Enter the number of jobs: ");
    scanf("%d", &n);
    int jobs[MAX_JOBS];
    printf("Enter the sizes of jobs:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &jobs[i]);
    }
    firstFit(blocks, m, jobs, n);
    return 0;
}
```

output:

```
Enter the number of memory blocks: 2
Enter the sizes of memory blocks:
5
2
Enter the number of jobs: 3
Enter the sizes of jobs:
1
1
1
Job Number      Job Size        Block Number
1               1               1
2               1               1
3               1               1
```

3) a) Shortest Job First (SJF) Scheduling algorithm:

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

```
struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int waiting_time;
    int turnaround_time;
};

void shortestJobFirst(struct Process processes[], int n) {
    int currentTime = 0;
    int completed = 0;
    int shortestProcess = -1;
    int shortestBurst = 999999; // Assuming a very large initial burst time
    while (completed != n) {
        shortestProcess = -1;
        shortestBurst = 999999;
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= currentTime && processes[i].burst_time < shortestBurst
&& processes[i].burst_time > 0) {
                shortestProcess = i;
                shortestBurst = processes[i].burst_time;
            }
        }
        if (shortestProcess == -1) {
            currentTime++;
```

```
        } else {
            processes[shortestProcess].burst_time--;
            currentTime++;
            if (processes[shortestProcess].burst_time == 0) {
                completed++;
                processes[shortestProcess].turnaround_time = currentTime -
processes[shortestProcess].arrival_time;
                processes[shortestProcess].waiting_time =
processes[shortestProcess].turnaround_time - processes[shortestProcess].burst_time;
            }}}
    printf("Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival_time,
               processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time);
    }
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];
    printf("Enter process details:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        processes[i].id = i + 1;
```

```
            printf("Arrival Time: ");

            scanf("%d", &processes[i].arrival_time);

            printf("Burst Time: ");

            scanf("%d", &processes[i].burst_time);

        }

        shortestJobFirst(processes, n);

        return 0;

}
```

output:

```
Enter the number of processes: 3
Enter process details:
Process 1:
Arrival Time: 0
Burst Time: 4
Process 2:
Arrival Time: 1
Burst Time: 4
Process 3:
Arrival Time: 2
Burst Time: 3
Process ID      Arrival Time    Burst Time      Waiting Time    Turnaround Time
1               0               0               4               4
2               1               0               10              10
3               2               0               5               5
```

4) a) paging concept using C program :

```
#include <stdio.h>

#include <stdlib.h>


#define PAGE_SIZE 4096

#define NUM_PAGES 256


struct PageTableEntry {

        int frameNumber;

        int validBit;

};
```

```
struct PageTableEntry pageTable[NUM_PAGES];

int main() {
	int logicalAddress;

	printf("Enter a logical address: ");
	scanf("%d", &logicalAddress);

	int pageNumber = logicalAddress / PAGE_SIZE;
	int offset = logicalAddress % PAGE_SIZE;

	if (pageNumber >= 0 && pageNumber < NUM_PAGES && pageTable[pageNumber].validBit) {
		int physicalAddress = pageTable[pageNumber].frameNumber * PAGE_SIZE + offset;
		printf("Physical Address: %d\n", physicalAddress);
	} else {
		printf("Page Fault!\n");
	}

	return 0;
}
```

output:

```
Enter a logical address: 123
Page Fault!
```

5) a) LFU page replacement algorithm using C program :

program:

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

```c
#include <stdlib.h>

#define NUM_FRAMES 3
#define NUM_PAGES 10

struct PageTableEntry {
    int pageNumber;
    int frequency;
};

struct Frame {
    int pageNumber;
    int frequency;
};

struct PageTableEntry pageTable[NUM_PAGES];
struct Frame frames[NUM_FRAMES];

int main() {
    // Initialize page table entries and frames
    for (int i = 0; i < NUM_PAGES; i++) {
        pageTable[i].pageNumber = i;
        pageTable[i].frequency = 0;
    }
    for (int i = 0; i < NUM_FRAMES; i++) {
```

```
        frames[i].pageNumber = -1;

        frames[i].frequency = 0;

    }


    int pageSequence[NUM_PAGES] = {0, 1, 2, 3, 0, 1, 4, 0, 1, 2}; // Sample page sequence


    int pageFaults = 0;


    // Simulate LFU page replacement algorithm

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

        int page = pageSequence[i];


        // Check if page is already in frames

        int found = 0;

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

            if (frames[j].pageNumber == page) {

                frames[j].frequency++;

                found = 1;

                break;

            }

        }


        // If page not found in frames, find the least frequently used frame

        if (!found) {

            int leastFreqIndex = 0;
```

```
                for (int j = 1; j < NUM_FRAMES; j++) {
                    if (frames[j].frequency < frames[leastFreqIndex].frequency) {
                        leastFreqIndex = j;
                    }
                }
                frames[leastFreqIndex].pageNumber = page;
                frames[leastFreqIndex].frequency = 1;
                pageFaults++;
            }

            // Update page table entry frequency
            pageTable[page].frequency++;
        }

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

        return 0;
}
```

output:

```
Number of Page Faults: 7
```

b)shell script to perform arithmetic operations:

program:

```
#!/bin/bash

echo "Enter two numbers:"
read num1
```

```
read num2

echo "Select an operation:"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read choice
case $choice in
    1)
        result=$(expr $num1 + $num2)
        echo "Result: $result"
        ;;
    2)
        result=$(expr $num1 - $num2)
        echo "Result: $result"
        ;;
    3)
        result=$(expr $num1 \* $num2)
        echo "Result: $result"
        ;;
    4)
        if [ $num2 -eq 0 ]; then
            echo "Error: Division by zero"
        else
            result=$(expr $num1 / $num2)
            echo "Result: $result"
        fi
```

```
        ;;
    *)
        echo "Invalid choice"
        ;;
esac
```

output:

```
Enter two numbers:
5
8
Select an operation:
1. Addition
2. Subtraction
3. Multiplication
4. Division
2
Result: -3
```

6) a)(LRU) page replacement algorithm:

```
#include <stdio.h>


#define NUM_FRAMES 3

#define NUM_PAGES 10


struct Frame {

    int pageNumber;

    int lastUsed;

};


struct Frame frames[NUM_FRAMES];


void initializeFrames() {
```

```c
    for (int i = 0; i < NUM_FRAMES; i++) {
        frames[i].pageNumber = -1;
        frames[i].lastUsed = 0;
    }
}

void displayFrames() {
    printf("Frames: ");
    for (int i = 0; i < NUM_FRAMES; i++) {
        if (frames[i].pageNumber == -1) {
            printf("[Empty] ");
        } else {
            printf("[%d] ", frames[i].pageNumber);
        }
    }
    printf("\n");
}

void updateLastUsed(int index) {
    frames[index].lastUsed++;
    for (int i = 0; i < NUM_FRAMES; i++) {
        if (i != index && frames[i].pageNumber != -1) {
            frames[i].lastUsed--;
        }
    }
```

```
}

int main() {
    int pageSequence[NUM_PAGES] = {0, 1, 2, 3, 0, 1, 4, 0, 1, 2}; // Sample page sequence

    initializeFrames();
    int pageFaults = 0;

    for (int i = 0; i < NUM_PAGES; i++) {
        int page = pageSequence[i];
        int found = 0;

        for (int j = 0; j < NUM_FRAMES; j++) {
            if (frames[j].pageNumber == page) {
                updateLastUsed(j);
                found = 1;
                break;
            }
        }

        if (!found) {
            int lruIndex = 0;
            for (int j = 1; j < NUM_FRAMES; j++) {
                if (frames[j].lastUsed < frames[lruIndex].lastUsed) {
                    lruIndex = j;
```

```
                }
            }
            frames[lruIndex].pageNumber = page;
            updateLastUsed(lruIndex);
            pageFaults++;
        }

        displayFrames();
    }

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

    return 0;
}
```

output:

```
Frames: [0] [Empty] [Empty]
Frames: [0] [1] [Empty]
Frames: [2] [1] [Empty]
Frames: [2] [3] [Empty]
Frames: [0] [3] [Empty]
Frames: [0] [1] [Empty]
Frames: [4] [1] [Empty]
Frames: [4] [0] [Empty]
Frames: [1] [0] [Empty]
Frames: [1] [2] [Empty]
Number of Page Faults: 10
```

b)shell script to find the greatest among three numbers:

```
#!/bin/bash

echo "Enter three numbers:"
```

```
read num1

read num2

read num3


if [ $num1 -gt $num2 ] && [ $num1 -gt $num3 ]; then

    echo "$num1 is the greatest."

elif [ $num2 -gt $num1 ] && [ $num2 -gt $num3 ]; then

    echo "$num2 is the greatest."

else

    echo "$num3 is the greatest."

fi
```

output:

```
Enter three numbers:
5
56
67
67 is the greatest.
```

7)a) First Come First Serve (FCFS) page replacement algorithm:

```
#include <stdio.h>

#include <stdlib.h>


#define NUM_FRAMES 3

#define NUM_PAGES 10


struct Frame {

    int pageNumber;

};
```

```
struct Frame frames[NUM_FRAMES];

void initializeFrames() {
    for (int i = 0; i < NUM_FRAMES; i++) {
        frames[i].pageNumber = -1;
    }
}

void displayFrames() {
    printf("Frames: ");
    for (int i = 0; i < NUM_FRAMES; i++) {
        if (frames[i].pageNumber == -1) {
            printf("[Empty] ");
        } else {
            printf("[%d] ", frames[i].pageNumber);
        }
    }
    printf("\n");
}

int main() {
    int pageSequence[NUM_PAGES] = {0, 1, 2, 3, 0, 1, 4, 0, 1, 2}; // Sample page sequence

    initializeFrames();
```

```
int pageFaults = 0;

for (int i = 0; i < NUM_PAGES; i++) {
    int page = pageSequence[i];
    int found = 0;

    for (int j = 0; j < NUM_FRAMES; j++) {
        if (frames[j].pageNumber == page) {
            found = 1;
            break;
        }
    }

    if (!found) {
        for (int j = 0; j < NUM_FRAMES; j++) {
            if (frames[j].pageNumber == -1) {
                frames[j].pageNumber = page;
                break;
            }
        }
        pageFaults++;
    }

    displayFrames();
}
```

```
    printf("Number of Page Faults: %d\n", pageFaults);

    return 0;
}
```

output:

```
Frames: [0] [Empty] [Empty]
Frames: [0] [1] [Empty]
Frames: [2] [1] [Empty]
Frames: [2] [3] [Empty]
Frames: [0] [3] [Empty]
Frames: [0] [1] [Empty]
Frames: [4] [1] [Empty]
Frames: [4] [0] [Empty]
Frames: [1] [0] [Empty]
Frames: [1] [2] [Empty]
Number of Page Faults: 10
```

8)a)Banker's algorithm for deadlock detection using C:

```
#include <stdio.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

int available[MAX_RESOURCES];
int max[MAX_PROCESSES][MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
int finish[MAX_PROCESSES];

int requestResources(int process, int request[]) {
```

```
for (int i = 0; i < MAX_RESOURCES; i++) {
    if (request[i] > need[process][i] || request[i] > available[i]) {
        return 0; // Request exceeds need or available resources
    }
}

// Pretend to allocate resources temporarily to check safety
for (int i = 0; i < MAX_RESOURCES; i++) {
    available[i] -= request[i];
    allocation[process][i] += request[i];
    need[process][i] -= request[i];
}

// Check for safety
int safeSequence[MAX_PROCESSES];
int work[MAX_RESOURCES];
for (int i = 0; i < MAX_RESOURCES; i++) {
    work[i] = available[i];
}

int count = 0;
while (count < MAX_PROCESSES) {
    int found = 0;
    for (int i = 0; i < MAX_PROCESSES; i++) {
        if (!finish[i]) {
```

```
            int j;
            for (j = 0; j < MAX_RESOURCES; j++) {
                if (need[i][j] > work[j]) {
                    break;
                }
            }
            if (j == MAX_RESOURCES) {
                for (int k = 0; k < MAX_RESOURCES; k++) {
                    work[k] += allocation[i][k];
                }
                safeSequence[count++] = i;
                finish[i] = 1;
                found = 1;
            }
        }
    }
    if (!found) {
        // Rollback changes
        for (int i = 0; i < MAX_RESOURCES; i++) {
            available[i] += request[i];
            allocation[process][i] -= request[i];
            need[process][i] += request[i];
        }
        return 0; // Unsafe state, request denied
    }
```

```c
    }

    // Grant the request and print the safe sequence
    printf("Safe Sequence: ");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("%d ", safeSequence[i]);
    }
    printf("\n");

    return 1; // Request granted
}

int main() {
    // Initialize available resources
    printf("Enter available resources:\n");
    for (int i = 0; i < MAX_RESOURCES; i++) {
        scanf("%d", &available[i]);
    }

    // Initialize maximum resources needed
    printf("Enter maximum resources needed for each process:\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("Process %d:\n", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            scanf("%d", &max[i][j]);
```

```c
        }
}

// Initialize allocation and need matrices
for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
                allocation[i][j] = 0;
                need[i][j] = max[i][j];
        }
        finish[i] = 0;
}

// Simulate resource requests
int process;
printf("Enter process number to request resources: ");
scanf("%d", &process);

int request[MAX_RESOURCES];
printf("Enter resource request for process %d:\n", process);
for (int i = 0; i < MAX_RESOURCES; i++) {
        scanf("%d", &request[i]);
}

if (requestResources(process, request)) {
        printf("Request granted.\n");
```

```
    } else {
        printf("Request denied.\n");
    }

    return 0;
}
```

output:

```
Enter available resources:
4
5
8
Enter maximum resources needed for each process:
Process 0:
3
1
2
Process 1:
1
1
1
Process 2:
02
1
3
Process 3:
1
1
1
Process 4:
2
3
1
Enter process number to request resources: 1 0 2
Enter resource request for process 1:
1
Request denied.
```

b) shell script to find the factorial of a given number:

```
#!/bin/bash

echo "Enter a number:"
read num
fact=1
for ((i = 1; i <= num; i++)); do
```

```
    fact=$((fact * i))
done
echo "Factorial of $num is $fact"
```

output:

```
Enter a number:
5
Factorial of 5 is 120
```

9)(FCFS) Scheduling algorithm:

```
#include <stdio.h>

struct Process {
    int id;
    int arrivalTime;
    int burstTime;
    int completionTime;
    int waitingTime;
    int turnaroundTime;
};

void calculateTimes(struct Process processes[], int n) {
    int currentTime = 0;

    for (int i = 0; i < n; i++) {
        if (currentTime < processes[i].arrivalTime) {
            currentTime = processes[i].arrivalTime;
        }
```

```c
        processes[i].completionTime = currentTime + processes[i].burstTime;

        processes[i].turnaroundTime = processes[i].completionTime - processes[i].arrivalTime;

        processes[i].waitingTime = processes[i].turnaroundTime - processes[i].burstTime;

        currentTime = processes[i].completionTime;

    }

}


void displayResults(struct Process processes[], int n) {

    printf("Process\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround Time\n");

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

        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrivalTime,

               processes[i].burstTime, processes[i].completionTime, processes[i].waitingTime,

               processes[i].turnaroundTime);

    }

}


int main() {

    int n;

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

    scanf("%d", &n);


    struct Process processes[n];


    printf("Enter process details:\n");

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

```c
            processes[i].id = i + 1;

            printf("Process %d:\n", processes[i].id);

            printf("Arrival Time: ");

            scanf("%d", &processes[i].arrivalTime);

            printf("Burst Time: ");

            scanf("%d", &processes[i].burstTime);

        }



        calculateTimes(processes, n);

        displayResults(processes, n);



        return 0;

}
```

output:

```
Enter the number of processes: 3
Enter process details:
Process 1:
Arrival Time: 0
Burst Time: 5
Process 2:
Arrival Time: 1
Burst Time: 4
Process 3:
Arrival Time: 5
Burst Time: 2
Process Arrival Time    Burst Time      Completion Time Waiting Time    Turnaround Time
1       0               5               5               0               5
2       1               4               9               4               8
3       5               2               11              4               6
```

10)

a) C program to implement Linked list   file allocation Strategy :

program:

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>
```

```
#define BLOCK_SIZE 512

struct File {
    char name[50];
    int size;
    struct File *next;
};

struct Block {
    int blockNumber;
    int size;
    struct File *file;
    struct Block *next;
};

struct Block *head = NULL;

void createBlock(int blockNumber, int size) {
    struct Block *newBlock = (struct Block *)malloc(sizeof(struct Block));
    newBlock->blockNumber = blockNumber;
    newBlock->size = size;
    newBlock->file = NULL;
    newBlock->next = NULL;

    if (head == NULL) {
        head = newBlock;
    } else {
        struct Block *temp = head;
```

```
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = newBlock;
        }
}

void allocateFile(char name[], int size) {
        struct File *newFile = (struct File *)malloc(sizeof(struct File));
        strcpy(newFile->name, name);
        newFile->size = size;
        newFile->next = NULL;

        struct Block *temp = head;
        while (temp != NULL) {
            if (temp->size >= size && temp->file == NULL) {
                temp->file = newFile;
                return;
            }
            temp = temp->next;
        }

        printf("Error: Not enough free space to allocate file.\n");
}

void displayBlocks() {
        struct Block *temp = head;
        while (temp != NULL) {
```

```
            printf("Block Number: %d, Size: %d\n", temp->blockNumber, temp->size);
            if (temp->file != NULL) {
                printf("    File Name: %s, File Size: %d\n", temp->file->name, temp->file->size);
            } else {
                printf("    Free Space\n");
            }
            temp = temp->next;
        }
    }

int main() {
        // Create blocks
        createBlock(1, 1024);
        createBlock(2, 512);
        createBlock(3, 768);

        // Allocate files
        allocateFile("file1.txt", 200);
        allocateFile("file2.txt", 400);
        allocateFile("file3.txt", 600);

        // Display blocks
        displayBlocks();

        return 0;
}
```

output:

```
Block Number: 1, Size: 1024
  File Name: file1.txt, File Size: 200
Block Number: 2, Size: 512
  File Name: file2.txt, File Size: 400
Block Number: 3, Size: 768
  File Name: file3.txt, File Size: 600
```

b) arithmetic operations:

```bash
#!/bin/bash

echo "Enter two numbers:"
read num1
read num2

echo "Select an operation:"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"

read choice

case $choice in
    1)
        result=$((num1 + num2))
        echo "Result: $result"
        ;;
    2)
        result=$((num1 - num2))
        echo "Result: $result"
        ;;
```

```
    3)
        result=$((num1 * num2))
        echo "Result: $result"
        ;;
    4)
        if [ $num2 -eq 0 ]; then
            echo "Error: Division by zero"
        else
            result=$((num1 / num2))
            echo "Result: $result"
        fi
        ;;
    *)
        echo "Invalid choice"
        ;;
esac
```

output:

```
Enter two numbers:
5
6
Select an operation:
1. Addition
2. Subtraction
3. Multiplication
4. Division
1
Result: 11
```

11) a)How the data is allocated sequentially, Write a C program to implement :

```
#include <stdio.h>


int main() {
```

```
    int array[5]; // Array to hold integers
    int i;

    // Input data into the array
    printf("Enter 5 integers:\n");
    for (i = 0; i < 5; i++) {
        scanf("%d", &array[i]);
    }

    // Display the array elements sequentially
    printf("Array elements:\n");
    for (i = 0; i < 5; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

output:

```
Enter 5 integers:
3
4
6
7
3
Array elements:
3 4 6 7 3
```

12)a) Write a C program to implement Producer-Consumer Problem using semaphore

concept:

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

```c
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5

sem_t empty, full, mutex;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

void *producer(void *arg) {
    int item = 1;
    while (1) {
        sem_wait(&empty);
        sem_wait(&mutex);

        // Produce item
        buffer[in] = item++;
        in = (in + 1) % BUFFER_SIZE;
        printf("Produced item %d\n", item - 1);

        sem_post(&mutex);
        sem_post(&full);

        sleep(1); // Sleep for simulation
    }
}

void *consumer(void *arg) {
```

```
    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);

        // Consume item
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("Consumed item %d\n", item);

        sem_post(&mutex);
        sem_post(&empty);

        sleep(1); // Sleep for simulation
    }
}

int main() {
    pthread_t prod_thread, cons_thread;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    // Create producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);
```

```
    // Wait for threads to finish
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    // Destroy semaphores
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

output:

```
};
```

```
Produced item 1
Consumed item 1
Produced item 2
Consumed item 2
Produced item 3
Consumed item 3
Produced item 4
Consumed item 4
Produced item 5
Consumed item 5
Produced item 6
Consumed item 6
Produced item 7
Consumed item 7
Produced item 8
Consumed item 8
Produced item 9
Consumed item 9
Produced item 10
Consumed item 10
Produced item 11
Consumed item 11
Produced item 12
Consumed item 12
Produced item 13
Consumed item 13
Produced item 14
Consumed item 14
```

13) a) Write a C program to implement First Round Robin Scheduling algorithm :

program:

```
#include <stdio.h>


struct Process {
    int id;
    int arrivalTime;
    int burstTime;
    int remainingTime;
};
```

```c
void roundRobinScheduling(struct Process processes[], int n, int timeQuantum) {
    int remainingProcesses = n;
    int currentTime = 0;
    int completedProcesses = 0;

    while (completedProcesses < n) {
        for (int i = 0; i < n; i++) {
            if (processes[i].remainingTime > 0) {
                if (processes[i].remainingTime > timeQuantum) {
                    currentTime += timeQuantum;
                    processes[i].remainingTime -= timeQuantum;
                } else {
                    currentTime += processes[i].remainingTime;
                    processes[i].remainingTime = 0;
                    processes[i].completionTime = currentTime;
                    completedProcesses++;
                }
            }
        }
    }
}

void displayResults(struct Process processes[], int n) {
    printf("Process\tArrival Time\tBurst Time\tCompletion Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrivalTime,
                  processes[i].burstTime, processes[i].completionTime);
    }
```

```c
}

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

    struct Process processes[n];

    printf("Enter time quantum for Round Robin: ");
    scanf("%d", &timeQuantum);

    printf("Enter process details:\n");
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Process %d:\n", processes[i].id);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrivalTime);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burstTime);
        processes[i].remainingTime = processes[i].burstTime;
    }

    roundRobinScheduling(processes, n, timeQuantum);
    displayResults(processes, n);

    return 0;
}
```

output:

```
Enter the number of processes: 4
Enter time quantum for Round Robin: 2
Enter process details:
Process 1:
Arrival Time: 0
Burst Time: 3
Process 2:
Arrival Time: 1
Burst Time: 4
Process 3:
Arrival Time: 2
Burst Time: 4
Process 4:
Arrival Time: 3
Burst Time: 9
Process Arrival Time    Burst Time      Completion Time
1       0               3               9
2       1               4               11
3       2               4               13
4       3               9               20
```

14)a. Implement pipe concept in Inter Process Communication using C program :

program:

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>


int main() {

    int pipefd[2];

    pid_t pid;

    char buffer[50];


    if (pipe(pipefd) == -1) {

        perror("pipe");

        exit(EXIT_FAILURE);

    }
```

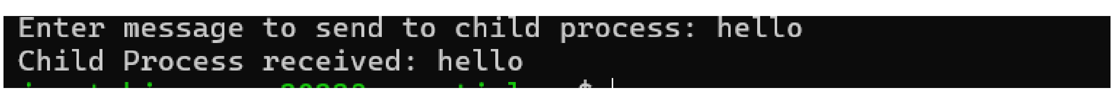
```c
    pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {    // Child process
        close(pipefd[1]);    // Close the write end of the pipe
        read(pipefd[0], buffer, sizeof(buffer));
        printf("Child Process received: %s", buffer);
        close(pipefd[0]);
    } else {    // Parent process
        close(pipefd[0]);    // Close the read end of the pipe
        printf("Enter message to send to child process: ");
        fgets(buffer, sizeof(buffer), stdin);
        write(pipefd[1], buffer, strlen(buffer) + 1);
        close(pipefd[1]);
    }

    return 0;
}
```

output:



14) a. Implement the concept of Threading and Synchronization using C Program:

program:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>


#define NUM_THREADS 5


int sharedVariable = 0;

pthread_mutex_t mutex;


void *threadFunction(void *arg) {

    int thread_id = *((int *)arg);


    pthread_mutex_lock(&mutex);

    sharedVariable++;

    printf("Thread %d: Incremented sharedVariable to %d\n", thread_id, sharedVariable);

    pthread_mutex_unlock(&mutex);


    pthread_exit(NULL);

}


int main() {

    pthread_t threads[NUM_THREADS];

    int thread_args[NUM_THREADS];


    pthread_mutex_init(&mutex, NULL);


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

        thread_args[i] = i + 1;
```

```c
        if (pthread_create(&threads[i], NULL, threadFunction, (void *)&thread_args[i]) != 0) {
            fprintf(stderr, "Error creating thread\n");
            exit(EXIT_FAILURE);
        }
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);

    return 0;
}
```

output:

```
Thread 1: Incremented sharedVariable to 1
Thread 2: Incremented sharedVariable to 2
Thread 3: Incremented sharedVariable to 3
Thread 4: Incremented sharedVariable to 4
Thread 5: Incremented sharedVariable to 5
```