Write a program to print the Child process ID and Parent process ID in both Child and Parent processes

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

int main() {
        pid_t pid;

        pid = fork(); // create a child process

        if (pid < 0) { // fork failed
        fprintf(stderr, "Fork failed.\n");
        return 1;
        } else if (pid == 0) { // child process
        printf("Child: My PID is %d. My parent's PID is %d.\n", getpid(), getppid());
        } else { // parent process
        printf("Parent: My PID is %d. My child's PID is %d.\n", getpid(), pid);
        }

        return 0;
}
```

C program to implement Producer-Consumer problem using semaphore

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

#define BUFFER_SIZE 5
#define NUM_ITEMS 10

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

sem_t empty;  // Counts the number of empty slots in the buffer
sem_t full;   // Counts the number of filled slots in the buffer
sem_t mutex;  // Provides mutual exclusion for buffer access
```

```
void* producer(void* arg) {
        int item;
        for (int i = 0; i < NUM_ITEMS; i++) {
        item = i;
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item;
        printf("Produced item: %d\n", item);
        in = (in + 1) % BUFFER_SIZE;
        sem_post(&mutex);
        sem_post(&full);
        }
        pthread_exit(NULL);
}

void* consumer(void* arg) {
        int item;
        for (int i = 0; i < NUM_ITEMS; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        item = buffer[out];
        printf("Consumed item: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;
        sem_post(&mutex);
        sem_post(&empty);
        }
        pthread_exit(NULL);
}

int main() {
        pthread_t producer_thread, consumer_thread;

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

        pthread_create(&producer_thread, NULL, producer, NULL);
        pthread_create(&consumer_thread, NULL, consumer, NULL);

        pthread_join(producer_thread, NULL);
```

```
    pthread_join(consumer_thread, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

Write a shell script to reverse and calculate the length of the string

```
#!/bin/bash

# Function to reverse a string
reverse_string() {
    local input_string=$1
    local reversed_string=""
    local length=${#input_string}

    for (( i=length-1; i>=0; i-- )); do
    reversed_string="$reversed_string${input_string:i:1}"
    done

    echo "$reversed_string"
}

# Function to calculate the length of a string
calculate_length() {
    local input_string=$1
    local length=${#input_string}
    echo "$length"
}

# Main script

# Read the input string from user
echo -n "Enter a string: "
read input

# Reverse the string
```

```
reversed=$(reverse_string "$input")
echo "Reversed string: $reversed"

# Calculate the length of the string
length=$(calculate_length "$input")
echo "Length of the string: $length"
```

Write a C program to implement process management using the following system calls fork, exec, getpid, exit, wait, close.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
        pid_t pid;

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

        if (pid < 0) { // Fork failed
        fprintf(stderr, "Fork failed.\n");
        return 1;
        } else if (pid == 0) { // Child process
        printf("Child Process\n");
        printf("Child PID: %d\n", getpid());
        printf("Child's Parent PID: %d\n", getppid());

        // Execute a command using exec
        char *args[] = {"ls", "-l", NULL};
        execvp(args[0], args);

        // Executed only if exec fails
        fprintf(stderr, "Exec failed.\n");
        return 1;
        } else { // Parent process
        printf("Parent Process\n");
        printf("Parent PID: %d\n", getpid());
        printf("Parent's Child PID: %d\n", pid);
```

```
        int status;
        wait(&status); // Wait for child process to finish

        if (WIFEXITED(status)) {
        printf("Child process exited with status: %d\n", WEXITSTATUS(status));
        }
        }

        return 0;
}
```

Write a program to send a message (pass through command line arguments) into a message queue. Send few messages with unique message numbers

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_MSG_SIZE 100

struct message {
        long type;
        char text[MAX_MSG_SIZE];
};

int main(int argc, char *argv[]) {
        if (argc < 2) {
        fprintf(stderr, "Usage: %s <message1> <message2> ...\n", argv[0]);
        return 1;
        }

        key_t key;
        int msgid;

        // Generate a unique key
        key = ftok(".", 'A');
        if (key == -1) {
```

```
        perror("ftok");
        return 1;
        }

        // Create a message queue or get the ID if it already exists
        msgid = msgget(key, 0666 | IPC_CREAT);
        if (msgid == -1) {
        perror("msgget");
        return 1;
        }

        // Send messages to the message queue
        for (int i = 1; i < argc; i++) {
        struct message msg;
        msg.type = i;
        snprintf(msg.text, MAX_MSG_SIZE, "%s", argv[i]);

        if (msgsnd(msgid, &msg, sizeof(struct message) - sizeof(long), 0) == -1) {
        perror("msgsnd");
        return 1;
        }

        printf("Sent message %d: %s\n", i, msg.text);
        }

        return 0;
}
```

Write a shell program to check whether the given string is palindrome or not

```
#!/bin/bash

# Function to check if a string is a palindrome
is_palindrome() {
        local input_string=$1
        local reversed_string=$(echo "$input_string" | rev)

        if [ "$input_string" = "$reversed_string" ]; then
        echo "The string '$input_string' is a palindrome."
        else
```

```
        echo "The string '$input_string' is not a palindrome."
        fi
}

# Main script

# Read the input string from user
echo -n "Enter a string: "
read input

# Remove spaces and convert to lowercase
input=$(echo "$input" | tr -d ' ' | tr '[:upper:]' '[:lower:]')

# Call the function to check if the string is a palindrome
is_palindrome "$input"
```

Write a Shell program to check whether the given number is Armstrong or Not

```
#!/bin/bash

# Function to check if a number is an Armstrong number
is_armstrong() {
        local number=$1
        local length=${#number}
        local sum=0

        for (( i=0; i<$length; i++ )); do
        digit=${number:i:1}
        power=$(( digit ** length ))
        sum=$(( sum + power ))
        done

        if [ $sum -eq $number ]; then
        echo "$number is an Armstrong number."
        else
        echo "$number is not an Armstrong number."
        fi
}

# Main script
```

```
# Read the input number from user
echo -n "Enter a number: "
read number

# Call the function to check if the number is an Armstrong number
is_armstrong "$number"
```

Write a shell script to reverse and calculate the length of the string

```
#!/bin/bash

# Function to reverse a string
reverse_string() {
        local input_string=$1
        local reversed_string=$(echo "$input_string" | rev)
        echo "$reversed_string"
}

# Function to calculate the length of a string
calculate_length() {
        local input_string=$1
        local length=${#input_string}
        echo "$length"
}

# Main script

# Read the input string from user
echo -n "Enter a string: "
read input

# Reverse the string
reversed=$(reverse_string "$input")
echo "Reversed string: $reversed"

# Calculate the length of the string
length=$(calculate_length "$input")
echo "Length of the string: $length"
```

Write a C program to implement anyone CPU scheduling algorithm.

```
#include <stdio.h>

#define MAX_PROCESSES 10

struct Process {
	int processId;
	int arrivalTime;
	int burstTime;
	int waitingTime;
	int turnaroundTime;
};

void calculateWaitingTime(struct Process *processes, int n) {
	int currentTime = 0;

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

	processes[i].waitingTime = currentTime - processes[i].arrivalTime;
	currentTime += processes[i].burstTime;
	}
}

void calculateTurnaroundTime(struct Process *processes, int n) {
	for (int i = 0; i < n; i++) {
	processes[i].turnaroundTime = processes[i].waitingTime +
processes[i].burstTime;
	}
}

void calculateAverageTimes(struct Process *processes, int n, float *avgWaitingTime,
float *avgTurnaroundTime) {
	int totalWaitingTime = 0;
	int totalTurnaroundTime = 0;

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

```
        totalWaitingTime += processes[i].waitingTime;
        totalTurnaroundTime += processes[i].turnaroundTime;
        }

        *avgWaitingTime = (float)totalWaitingTime / n;
        *avgTurnaroundTime = (float)totalTurnaroundTime / n;
}

void displayProcessTable(struct Process *processes, int n) {
        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].processId,
processes[i].arrivalTime,
                processes[i].burstTime, processes[i].waitingTime,
processes[i].turnaroundTime);
        }
}

int main() {
        struct Process processes[MAX_PROCESSES];
        int n;

        printf("Enter the number of processes (up to %d): ", MAX_PROCESSES);
        scanf("%d", &n);

        printf("Enter the arrival time and burst time for each process:\n");
        for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrivalTime);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burstTime);
        processes[i].processId = i + 1;
        }

        calculateWaitingTime(processes, n);
        calculateTurnaroundTime(processes, n);

        float avgWaitingTime, avgTurnaroundTime;
```

```
    calculateAverageTimes(processes, n, &avgWaitingTime, &avgTurnaroundTime);

    displayProcessTable(processes, n);
    printf("Average Waiting Time: %.2f\n", avgWaitingTime);
    printf("Average Turnaround Time: %.2f\n", avgTurnaroundTime);

    return 0;
}
```

Write a C program to implement anyone Disk Scheduling algorithm.

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

#define MAX_REQUESTS 100

void calculateSeekTime(int requests[], int n, int initialPosition) {
    int totalSeekTime = 0;
    int currentPosition = initialPosition;
    int visited[MAX_REQUESTS] = {0};

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

    // Find the closest unvisited request
    for (int j = 0; j < n; j++) {
    if (!visited[j] && abs(requests[j] - currentPosition) < minDistance) {
        minDistance = abs(requests[j] - currentPosition);
        nextRequest = j;
    }
    }

    // Update the seek time and current position
    totalSeekTime += minDistance;
    currentPosition = requests[nextRequest];
    visited[nextRequest] = 1;
    }
```

```
    printf("Total Seek Time: %d\n", totalSeekTime);
}

int main() {
    int requests[MAX_REQUESTS];
    int n;
    int initialPosition;

    printf("Enter the number of disk requests (up to %d): ", MAX_REQUESTS);
    scanf("%d", &n);

    printf("Enter the disk requests:\n");
    for (int i = 0; i < n; i++) {
    printf("Request %d: ", i + 1);
    scanf("%d", &requests[i]);
    }

    printf("Enter the initial position of the disk head: ");
    scanf("%d", &initialPosition);

    calculateSeekTime(requests, n, initialPosition);

    return 0;
}
```