Write a shell script to produce a multiplication table.

```
#!/bin/bash

echo "Enter the number for the multiplication table:"
read number
echo "Enter the range (e.g., 1 to 10):"
read range
echo "Multiplication table for $number in the range $range:"
echo "----------------------------------------"
for i in $(seq $range); do
   result=$((number * i))
   echo "$number x $i = $result"
done
```

Write a shell script program to implement a small calculator.

```
#!/bin/bash

echo "Simple Calculator"
echo "-----------------"
echo "Enter first number:"
read num1
echo "Enter second number:"
read num2
echo "Select operation:"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read choice
case $choice in
```

```
    1) result=$((num1 + num2))
       operator="+";;
    2) result=$((num1 - num2))
       operator="-";;
    3) result=$((num1 * num2))
       operator="*";;
    4) result=$(awk "BEGIN {printf \"%.2f\", $num1 / $num2}")
       operator="/";;
    *) echo "Invalid choice"
       exit 1;;
esac
echo "Result: $num1 $operator $num2 = $result"
```

Write a shell script to display prime numbers up to the given limit

```
#!/bin/bash

echo "Enter the limit to find prime numbers up to:"
read limit
echo "Prime numbers up to $limit are:"
echo "2"
for ((num = 3; num <= limit; num += 2)); do
    is_prime=1

    for ((i = 2; i <= num / 2; i++)); do
        if [ $((num % i)) -eq 0 ]; then
            is_prime=0
            break
        fi
    done
```

```
    if [ $is_prime -eq 1 ]; then
        echo "$num"
    fi
done
```

Write a program using system calls to copy half the content of the file to a newly created file. (a) First half of the file (b) Second half of the file

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

void copyHalfFile(const char *inputFileName, const char *outputFileName, int isFirstHalf) {
    // Open the input file
    int inputFile = open(inputFileName, O_RDONLY);
    if (inputFile == -1) {
        perror("Error opening input file");
        exit(EXIT_FAILURE);
    }
 // Get the size of the file
    off_t fileSize = lseek(inputFile, 0, SEEK_END);
    lseek(inputFile, 0, SEEK_SET);
 // Calculate the position to split the file
    off_t splitPosition = isFirstHalf ? fileSize / 2 : fileSize;
 // Open the output file
    int outputFile = open(outputFileName, O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (outputFile == -1) {
        perror("Error opening output file");
        close(inputFile);
        exit(EXIT_FAILURE);
```

```
    }    // Copy the data
    char buffer[4096];
    ssize_t bytesRead, bytesWritten;
    while ((bytesRead = read(inputFile, buffer, sizeof(buffer))) > 0) {
        if (lseek(inputFile, 0, SEEK_CUR) >= splitPosition) {
            // If we've reached the split position, break out of the loop
            break;
        }
   bytesWritten = write(outputFile, buffer, bytesRead);
        if (bytesWritten != bytesRead) {
            perror("Error writing to output file");
            close(inputFile);
            close(outputFile);
            exit(EXIT_FAILURE);
        }
    }
  // Close the files
    close(inputFile);
    close(outputFile);
}
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <input_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
char *inputFileName = argv[1];
 // Copy the first half of the file to a new file
    copyHalfFile(inputFileName, "first_half.txt", 1);
    printf("First half copied to first_half.txt\n");

    // Copy the second half of the file to a new file
```

```
    copyHalfFile(inputFileName, "second_half.txt", 0);
    printf("Second half copied to second_half.txt\n");
   return 0;}
```

to compile

```
gcc copy_half_file.c -o copy_half_file
./copy_half_file input_file.txt
```

Write a program using system call to read from console until user enters '$' and print the same on a file

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#define MAX_BUFFER_SIZE 4096
int main() {
    char buffer[MAX_BUFFER_SIZE];
    char filename[] = "user_input.txt";
    // Open the file for writing
    int outputFile = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (outputFile == -1) {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }
    printf("Enter text (Enter '$' to stop):\n");
    ssize_t bytesRead;
    do {
        bytesRead = read(STDIN_FILENO, buffer, sizeof(buffer));
        if (bytesRead == -1) {
            perror("Error reading from console");
            close(outputFile);
```

```
            exit(EXIT_FAILURE);
        }
        // Write the buffer to the file
        ssize_t bytesWritten = write(outputFile, buffer, bytesRead);
        if (bytesWritten == -1) {
            perror("Error writing to file");
            close(outputFile);
            exit(EXIT_FAILURE);
        }
    } while (buffer[0] != '$');
    printf("Text entered by the user has been saved to %s\n", filename);
    // Close the file
    close(outputFile);
    return 0;
}
```

To compile

```
gcc read_console_and_save.c -o read_console_and_save
./read_console_and_save
```

Write a program using system call to read the contents of a file without using char array and display the contents on the console. (Do not use any built in functions like sizeof() and strlen())

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

#define BUFFER_SIZE 1

int main(int argc, char *argv[]) {
    if (argc != 2) {
```

```
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int inputFile = open(argv[1], O_RDONLY);
    if (inputFile == -1) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    char buffer[BUFFER_SIZE];
    ssize_t bytesRead;
    while ((bytesRead = read(inputFile, buffer, BUFFER_SIZE)) > 0) {
        // Print each character to the console
        for (ssize_t i = 0; i < bytesRead; i++) {
            write(STDOUT_FILENO, &buffer[i], 1);
        }
    }
    if (bytesRead == -1) {
        perror("Error reading from file");
        close(inputFile);
        exit(EXIT_FAILURE);
    }

    close(inputFile);
    return 0;
}
```

To compile

gcc read_file_without_char_array.c -o read_file_without_char_array

./read_file_without_char_array filename.txt

Write a program using system calls for operation on process to simulate that n fork calls create (2 n – 1) child processes.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
void simulateForks(int n) {
    for (int i = 0; i < n; i++) {
        pid_t child_pid = fork();
  if (child_pid == -1) {
            perror("Error creating child process");
            exit(EXIT_FAILURE);
        }
        if (child_pid == 0) {
            // Child process
            printf("Child process %d (PID: %d) created\n", i + 1, getpid());
            break;
        }
    }
}

int main() {
    int n;
    printf("Enter the value of n: ");
    scanf("%d", &n);
    if (n < 1) {
        fprintf(stderr, "Invalid value for n. It should be greater than or equal to 1.\n");
        exit(EXIT_FAILURE);
    }
    simulateForks(n);
    // Let the parent process wait for a while
    sleep(5);
```

```
    return 0;
}
```

To compile

```
gcc simulate_forks.c -o simulate_forks
./simulate_forks
```

. Write a program using systems for operations on processes to create a hierarchy of processes P1 → P2 → P3. Also print the id and parent id for each process.

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

int main() {
    // Process P1
    pid_t p1 = getpid();
    printf("P1 (PID: %d) - Parent ID: %d\n", p1, getppid());

    // Fork to create Process P2
    pid_t p2 = fork();

    if (p2 == -1) {
        perror("Error creating P2");
        exit(EXIT_FAILURE);
    }

    if (p2 == 0) {
        // Process P2
        printf("P2 (PID: %d) - Parent ID: %d\n", getpid(), getppid());

        // Fork to create Process P3
```

```
        pid_t p3 = fork();

        if (p3 == -1) {
            perror("Error creating P3");
            exit(EXIT_FAILURE);
        }

        if (p3 == 0) {
            // Process P3
            printf("P3 (PID: %d) - Parent ID: %d\n", getpid(), getppid());
        }
    }

    // Let the parent process wait for a while
    sleep(5);

    return 0;
}
```

To compile

gcc process_hierarchy.c -o process_hierarchy

./process_hierarchy

Write a program using system calls for operation on processes to create a hierarchy of processes as given in figure 1. Also, simulate process p4 as orphan and P5 as zombie. P1 P2 P4 P3 P5

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

```
int main() {
    // Process P1
    pid_t p1 = getpid();
    printf("P1 (PID: %d) - Parent ID: %d\n", p1, getppid());

    // Fork to create Process P2
    pid_t p2 = fork();

    if (p2 == -1) {
        perror("Error creating P2");
        exit(EXIT_FAILURE);
    }

    if (p2 == 0) {
        // Process P2
        printf("P2 (PID: %d) - Parent ID: %d\n", getpid(), getppid());

        // Fork to create Process P3
        pid_t p3 = fork();

        if (p3 == -1) {
            perror("Error creating P3");
            exit(EXIT_FAILURE);
        }

        if (p3 == 0) {
            // Process P3
            printf("P3 (PID: %d) - Parent ID: %d\n", getpid(), getppid());
        }
    } else {
        // Parent process (P1)
```

```
// Fork to create Process P4 (Orphan)
pid_t p4 = fork();

if (p4 == -1) {
    perror("Error creating P4");
    exit(EXIT_FAILURE);
}

if (p4 == 0) {
    // Process P4
    printf("P4 (PID: %d) - Parent ID: %d\n", getpid(), getppid());

    // Sleep to become orphan
    sleep(2);
    printf("P4 (PID: %d) - New Parent ID: %d (becomes orphan)\n", getpid(), getppid());
} else {
    // Parent process (P1) continues

    // Fork to create Process P5 (Zombie)
    pid_t p5 = fork();

    if (p5 == -1) {
        perror("Error creating P5");
        exit(EXIT_FAILURE);
    }

    if (p5 == 0) {
        // Process P5
        printf("P5 (PID: %d) - Parent ID: %d\n", getpid(), getppid());
```

```
            // Exit immediately to become a zombie
            exit(0);
        } else {
            // Parent process (P1) continues

            // Sleep to allow P4 to become orphan and P5 to become a zombie
            sleep(5);

            // Wait for P4 to exit
            waitpid(p4, NULL, 0);

            // Print status of P5 (should be a zombie)
            printf("Status of P5: Zombie (check with 'ps aux | grep P5')\n");
        }
    }
  }

  return 0;
}
```

To compile

```
gcc process_hierarchy_orphan_zombie.c -o process_hierarchy_orphan_zombie
./process_hierarchy_orphan_zombie
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
  // Process P1
  pid_t p1 = getpid();
```

```
printf("P1 (PID: %d) - Parent ID: %d\n", p1, getppid());
// Fork to create Process P2
pid_t p2 = fork();
if (p2 == -1) {
    perror("Error creating P2");
    exit(EXIT_FAILURE);
}
if (p2 == 0) {
    // Process P2
    printf("P2 (PID: %d) - Parent ID: %d\n", getpid(), getppid());
    // Fork to create Process P3
    pid_t p3 = fork();
    if (p3 == -1) {
        perror("Error creating P3");
        exit(EXIT_FAILURE);
    }
    if (p3 == 0) {
        // Process P3
        printf("P3 (PID: %d) - Parent ID: %d\n", getpid(), getppid());
        // Fork to create Process P6
        pid_t p6 = fork();
        if (p6 == -1) {
            perror("Error creating P6");
            exit(EXIT_FAILURE);
        }
        if (p6 == 0) {
            // Process P6
            printf("P6 (PID: %d) - Parent ID: %d\n", getpid(), getppid());
        }
    } else {
        // Parent process (P2)
```

```
        // Fork to create Process P5
        pid_t p5 = fork();
        if (p5 == -1) {
            perror("Error creating P5");
            exit(EXIT_FAILURE);
        }
        if (p5 == 0) {
            // Process P5
            printf("P5 (PID: %d) - Parent ID: %d\n", getpid(), getppid());
        }
    }
} else {
    // Parent process (P1)
    // Fork to create Process P4 (Orphan)
    pid_t p4 = fork();
    if (p4 == -1) {
        perror("Error creating P4");
        exit(EXIT_FAILURE);
    }
    if (p4 == 0) {
        // Process P4
        printf("P4 (PID: %d) - Parent ID: %d\n", getpid(), getppid());
        // Sleep to become orphan
        sleep(2);
        printf("P4 (PID: %d) - New Parent ID: %d (becomes orphan)\n", getpid(), getppid());
    } else {
        // Parent process (P1) continues
        // Fork to create Process P7 (Zombie)
        pid_t p7 = fork();
        if (p7 == -1) {
            perror("Error creating P7");
```

```
            exit(EXIT_FAILURE);
        }
        if (p7 == 0) {
            // Process P7
            printf("P7 (PID: %d) - Parent ID: %d\n", getpid(), getppid());
            // Exit immediately to become a zombie
            exit(0);
        } else {
            // Parent process (P1) continues
            // Sleep to allow P4 to become orphan and P7 to become a zombie
            sleep(5);
            // Wait for P4 to exit
            waitpid(p4, NULL, 0);
            // Print status of P7 (should be a zombie)
            printf("Status of P7: Zombie (check with 'ps aux | grep P7')\n");
        }
    }
  }
  return 0;
}
```

To compile

gcc process_hierarchy_orphan_zombie_2.c -o process_hierarchy_orphan_zombie_2

./process_hierarchy_orphan_zombie_2

Write a program using pthread to concatenate the strings, where multiple strings are passed to thread function.

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

```
#define MAX_STRING_LENGTH 100

// Structure to pass data to the thread function
struct ThreadData {
    char** strings;
    int numStrings;
    char result[MAX_STRING_LENGTH];
};

// Thread function to concatenate strings
void* concatenateStrings(void* arg) {
    struct ThreadData* threadData = (struct ThreadData*)arg;

    // Initialize the result string
    strcpy(threadData->result, "");

    // Concatenate the strings
    for (int i = 0; i < threadData->numStrings; ++i) {
        strcat(threadData->result, threadData->strings[i]);
    }

    pthread_exit(NULL);
}

int main() {
    // Example strings
    char* strings[] = {"Hello, ", "world", "!", "\n"};

    // Number of strings
    int numStrings = sizeof(strings) / sizeof(strings[0]);
```

```
    // Create thread data
    struct ThreadData threadData;
    threadData.strings = strings;
    threadData.numStrings = numStrings;

    // Create a pthread
    pthread_t thread;

    // Create the thread
    if (pthread_create(&thread, NULL, concatenateStrings, (void*)&threadData) != 0) {
        fprintf(stderr, "Error creating thread\n");
        exit(EXIT_FAILURE);
    }

    // Wait for the thread to finish
    if (pthread_join(thread, NULL) != 0) {
        fprintf(stderr, "Error joining thread\n");
        exit(EXIT_FAILURE);
    }

    // Print the concatenated string
    printf("Concatenated String: %s", threadData.result);

    return 0;
}
```

To compile

gcc concatenate_strings.c -o concatenate_strings -pthread

./concatenate_strings

Write a program using pthread to find the length of string, where strings are passed to thread function.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#define MAX_STRING_LENGTH 100
// Structure to pass data to the thread function
struct ThreadData {
   char** strings;
   int* lengths;
   int numStrings;
};
// Thread function to find the length of strings
void* findStringLengths(void* arg) {
   struct ThreadData* threadData = (struct ThreadData*)arg;
   // Calculate the length of each string
   for (int i = 0; i < threadData->numStrings; ++i) {
      threadData->lengths[i] = strlen(threadData->strings[i]);
   }
   pthread_exit(NULL);
}
int main() {
   // Example strings
   char* strings[] = {"Hello", "world", "!", "This", "is", "a", "test"};
   // Number of strings
   int numStrings = sizeof(strings) / sizeof(strings[0]);
   // Create thread data
   struct ThreadData threadData;
   threadData.strings = strings;
   threadData.lengths = (int*)malloc(numStrings * sizeof(int));
```

```
    threadData.numStrings = numStrings;
    // Create a pthread
    pthread_t thread;
    // Create the thread
    if (pthread_create(&thread, NULL, findStringLengths, (void*)&threadData) != 0) {
        fprintf(stderr, "Error creating thread\n");
        exit(EXIT_FAILURE);
    }
    // Wait for the thread to finish
    if (pthread_join(thread, NULL) != 0) {
        fprintf(stderr, "Error joining thread\n");
        exit(EXIT_FAILURE);
    }
    // Print the lengths of strings
    printf("String Lengths:\n");
    for (int i = 0; i < numStrings; ++i) {
        printf("%s: %d\n", threadData.strings[i], threadData.lengths[i]);
    }
    // Free allocated memory
    free(threadData.lengths);
    return 0;
}
```

To Compile

```
gcc string_lengths.c -o string_lengths -pthread
./string_lengths
```

Write a program that performs statistical operations of calculating the average, maximum and minimum for a set of numbers. Create three threads where each performs their respective operations.

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

```
#define NUM_THREADS 3
#define ARRAY_SIZE 10
// Structure to pass data to the thread functions
struct ThreadData {
    int* numbers;
    int arraySize;
    double average;
    int maximum;
    int minimum;
};
// Thread function to calculate average
void* calculateAverage(void* arg) {
    struct ThreadData* threadData = (struct ThreadData*)arg;
    // Calculate average
    double sum = 0.0;
    for (int i = 0; i < threadData->arraySize; ++i) {
        sum += threadData->numbers[i];
    }
    threadData->average = sum / threadData->arraySize;
    pthread_exit(NULL);
}
// Thread function to find the maximum value
void* findMaximum(void* arg) {
    struct ThreadData* threadData = (struct ThreadData*)arg;
    // Find maximum
    threadData->maximum = threadData->numbers[0];
    for (int i = 1; i < threadData->arraySize; ++i) {
        if (threadData->numbers[i] > threadData->maximum) {
            threadData->maximum = threadData->numbers[i];
        }
    }
```

```
    pthread_exit(NULL);
}
// Thread function to find the minimum value
void* findMinimum(void* arg) {
    struct ThreadData* threadData = (struct ThreadData*)arg;
    // Find minimum
    threadData->minimum = threadData->numbers[0];
    for (int i = 1; i < threadData->arraySize; ++i) {
        if (threadData->numbers[i] < threadData->minimum) {
            threadData->minimum = threadData->numbers[i];
        }
    }
    pthread_exit(NULL);
}
int main() {
    // Example numbers
    int numbers[ARRAY_SIZE] = {12, 5, 8, 20, 14, 7, 3, 16, 10, 18};
    // Create thread data
    struct ThreadData threadData;
    threadData.numbers = numbers;
    threadData.arraySize = ARRAY_SIZE;
    // Create threads
    pthread_t threads[NUM_THREADS];
    pthread_create(&threads[0], NULL, calculateAverage, (void*)&threadData);
    pthread_create(&threads[1], NULL, findMaximum, (void*)&threadData);
    pthread_create(&threads[2], NULL, findMinimum, (void*)&threadData);
    // Wait for threads to finish
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], NULL);
    }
    // Print results
```

```
    printf("Average: %.2f\n", threadData.average);
    printf("Maximum: %d\n", threadData.maximum);
    printf("Minimum: %d\n", threadData.minimum);
    return 0;
}
```

TO Compile

```
gcc statistical_operations.c -o statistical_operations -pthread
./statistical_operations
```

Write a multithreaded program where an array of integers is passed globally and is divided into two smaller lists and given as input to two threads. The thread will sort their half of the list and will pass the sorted list to a third thread which merges and sorts the list. The final sorted list is printed by the parent thread

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

#define ARRAY_SIZE 10

// Structure to pass data to the thread functions
struct ThreadData {
    int* numbers;
    int start;
    int end;
};

// Function to merge two sorted arrays
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
```

```
int n2 = r - m;

// Create temporary arrays
int L[n1], R[n2];

// Copy data to temporary arrays L[] and R[]
for (i = 0; i < n1; i++)
   L[i] = arr[l + i];
for (j = 0; j < n2; j++)
   R[j] = arr[m + 1 + j];

// Merge the temporary arrays back into arr[l..r]
i = 0;     // Initial index of first subarray
j = 0;     // Initial index of second subarray
k = l;     // Initial index of merged subarray
while (i < n1 && j < n2) {
   if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
   } else {
      arr[k] = R[j];
      j++;
   }
   k++;
}

// Copy the remaining elements of L[], if there are any
while (i < n1) {
   arr[k] = L[i];
   i++;
   k++;
```

```
    }

    // Copy the remaining elements of R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Thread function to sort a portion of the array
void* sortArray(void* arg) {
    struct ThreadData* threadData = (struct ThreadData*)arg;
    int* arr = threadData->numbers;
    int start = threadData->start;
    int end = threadData->end;

    // Base case: If the portion has only one element, it is already sorted
    if (start < end) {
        // Calculate mid point
        int mid = start + (end - start) / 2;

        // Recursively sort the first and second halves
        struct ThreadData data1 = {arr, start, mid};
        struct ThreadData data2 = {arr, mid + 1, end};

        pthread_t thread1, thread2;
        pthread_create(&thread1, NULL, sortArray, (void*)&data1);
        pthread_create(&thread2, NULL, sortArray, (void*)&data2);

        pthread_join(thread1, NULL);
```

```
        pthread_join(thread2, NULL);

        // Merge the sorted halves
        merge(arr, start, mid, end);
    }

    pthread_exit(NULL);
}

int main() {
    // Example array of integers
    int numbers[ARRAY_SIZE] = {12, 5, 8, 20, 14, 7, 3, 16, 10, 18};

    // Create thread data for the entire array
    struct ThreadData threadData = {numbers, 0, ARRAY_SIZE - 1};

    // Create a pthread for sorting the entire array
    pthread_t sortThread;
    pthread_create(&sortThread, NULL, sortArray, (void*)&threadData);

    // Wait for the sorting thread to finish
    pthread_join(sortThread, NULL);
    // Print the final sorted list
    printf("Sorted List: ");
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        printf("%d ", numbers[i]);
    }
    printf("\n");
    return 0;
}
```

TO Compile

gcc parallel_merge_sort.c -o parallel_merge_sort -pthread

./parallel_merge_sort

Implement the producer consumer problem using pthreads and mutex operations.

Test Cases: (a) A producer only produces if buffer is empty and consumer only consumes if some content is in the buffer.

(b) A producer produces(writes) an item in the buffer and consumer consumes(deletes) the last produces item in the buffer.

(c) A producer produces(writes) on the last consumed(deleted) index of the buffer.

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>


#define BUFFER_SIZE 5


// Shared buffer

int buffer[BUFFER_SIZE];

int count = 0;  // Number of items in the buffer

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t notEmpty = PTHREAD_COND_INITIALIZER;

pthread_cond_t notFull = PTHREAD_COND_INITIALIZER;


// Function prototypes

void* producer(void* arg);

void* consumer(void* arg);


int main() {

   // Initialize buffer to all zeros

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

      buffer[i] = 0;
```

```
    }

    // Create pthreads for producer and consumer
    pthread_t producerThread, consumerThread;
    pthread_create(&producerThread, NULL, producer, NULL);
    pthread_create(&consumerThread, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(producerThread, NULL);
    pthread_join(consumerThread, NULL);

    return 0;
}

// Producer thread function
void* producer(void* arg) {
    for (int i = 1; i <= BUFFER_SIZE; ++i) {
        pthread_mutex_lock(&mutex);

        // Test Case (a): Producer only produces if the buffer is empty
        while (count > 0) {
            pthread_cond_wait(&notFull, &mutex);
        }

        // Test Case (b): Producer produces an item in the buffer
        buffer[count] = i;
        printf("Produced: %d\n", i);
        count++;

        pthread_cond_signal(&notEmpty);
        pthread_mutex_unlock(&mutex);
```

```
    }

    pthread_exit(NULL);
}

// Consumer thread function
void* consumer(void* arg) {
    for (int i = 1; i <= BUFFER_SIZE; ++i) {
        pthread_mutex_lock(&mutex);

        // Test Case (a): Consumer only consumes if some content is in the buffer
        while (count == 0) {
            pthread_cond_wait(&notEmpty, &mutex);
        }

        // Test Case (b): Consumer consumes the last produced item in the buffer
        int consumed = buffer[count - 1];
        printf("Consumed: %d\n", consumed);
        count--;
        pthread_cond_signal(&notFull);
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
```

To compile

```
gcc producer_consumer.c -o producer_consumer -pthread
./producer_consumer
```

Implement the reader writer problem using semaphore and mutex operations to synchronize n readers active in reader section at a same time, and one writer active at a time.

Test Cases: (a) If n readers are active no writer is allowed to write.

(b) If one writer is writing no other writer should be allowed to read or write on the shared variable.

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

#define NUM_READERS 3
#define NUM_WRITERS 2

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t readerSemaphore, writerSemaphore;

int sharedVariable = 0;
int readerCount = 0;

// Function prototypes
void* reader(void* arg);
void* writer(void* arg);

int main() {
    // Initialize semaphores
    sem_init(&readerSemaphore, 0, NUM_READERS);
    sem_init(&writerSemaphore, 0, 1);

    // Create pthreads for readers and writers
    pthread_t readerThreads[NUM_READERS];
    pthread_t writerThreads[NUM_WRITERS];
```

```
    for (int i = 0; i < NUM_READERS; ++i) {
        pthread_create(&readerThreads[i], NULL, reader, NULL);
    }

    for (int i = 0; i < NUM_WRITERS; ++i) {
        pthread_create(&writerThreads[i], NULL, writer, NULL);
    }

    // Wait for threads to finish
    for (int i = 0; i < NUM_READERS; ++i) {
        pthread_join(readerThreads[i], NULL);
    }

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

    // Destroy semaphores
    sem_destroy(&readerSemaphore);
    sem_destroy(&writerSemaphore);

    return 0;
}

// Reader thread function
void* reader(void* arg) {
    while (1) {
        sem_wait(&readerSemaphore);  // Wait if maximum number of readers reached
        pthread_mutex_lock(&mutex);

        // Test Case (a): If n readers are active, no writer is allowed to write
```

```
        if (readerCount == 0) {
            sem_wait(&writerSemaphore);
        }

        readerCount++;
        pthread_mutex_unlock(&mutex);
        sem_post(&readerSemaphore);

        // Reading the shared variable
        printf("Reader %ld reads: %d\n", pthread_self(), sharedVariable);

        pthread_mutex_lock(&mutex);
        readerCount--;

        // Test Case (a): If n readers are active, release the writer semaphore
        if (readerCount == 0) {
            sem_post(&writerSemaphore);
        }

        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(NULL);
}

// Writer thread function
void* writer(void* arg) {
    while (1) {
        sem_wait(&writerSemaphore);  // Wait if another writer is active

        // Test Case (b): If one writer is writing, no other writer should be allowed to read or write
```

```
        sem_wait(&readerSemaphore);

        // Writing to the shared variable
        sharedVariable++;
        printf("Writer %ld writes: %d\n", pthread_self(), sharedVariable);

        sem_post(&readerSemaphore);
        sem_post(&writerSemaphore);
    }

    pthread_exit(NULL);
}
```

To Compile

```
gcc reader_writer.c -o reader_writer -pthread -lrt
./reader_writer
```