OPERATING SYSTEMS LAB - PRACTICAL 7 - THREADS

Name - Sakshi Soni
Roll No - 13

AIM -

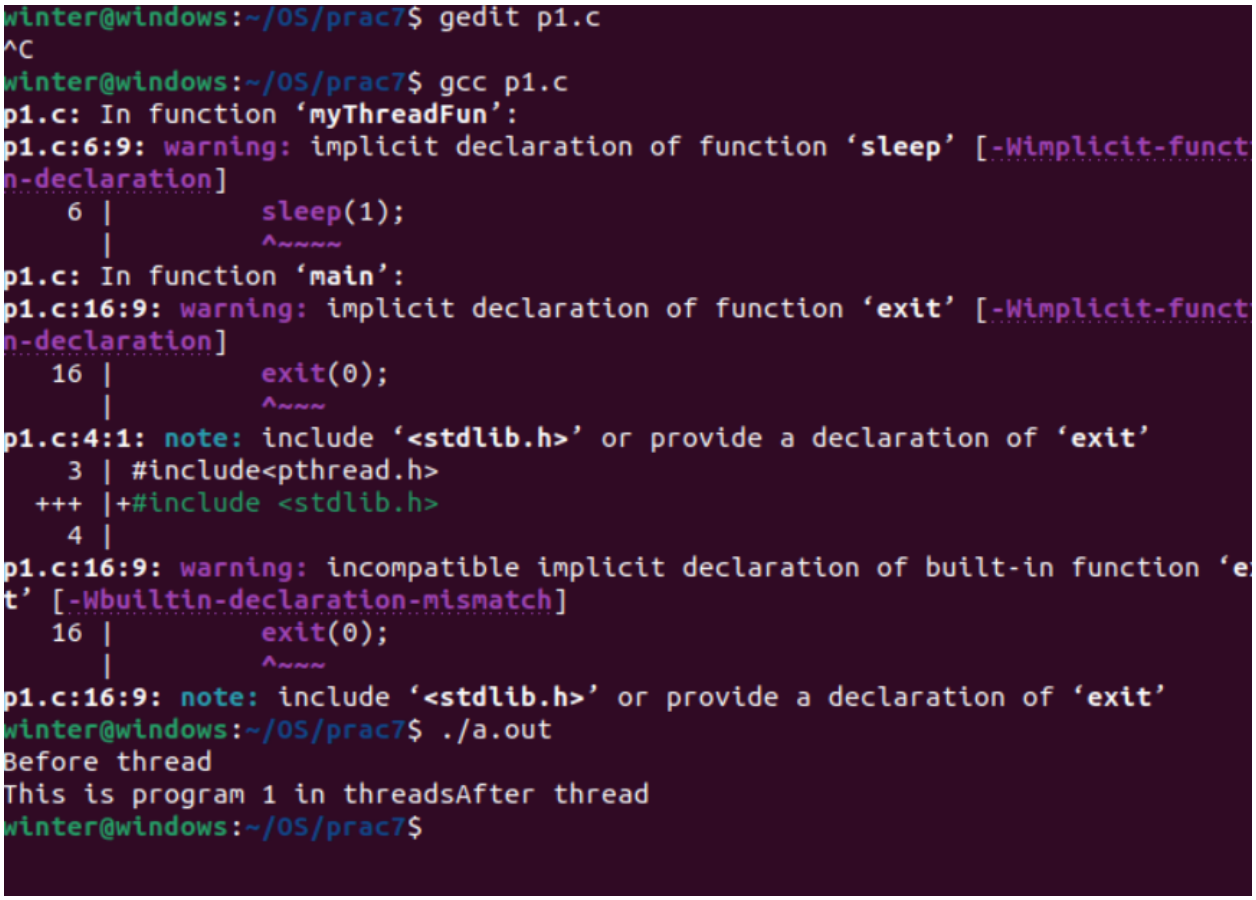
Write C programs to implement threads and semaphores for process synchronization

PROGRAM AND OUTPUT -

Program 1- To demonstrate thread system calls

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>


void *myThreadFun(void *vargp)

{

   sleep(1);

   printf("Printing GeeksQuiz from Thread \n");

   return NULL;

}


int main()

{
```

```
    pthread_t tid;
    printf("Before Thread\n");
    pthread_create(&tid, NULL, myThreadFun, NULL);
    pthread_join(tid, NULL);
    printf("After Thread\n");
    exit(0);
}
```



Program 2-
Implement multiple threads with global and static variables

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

```
#include <stdlib.h>
#include <pthread.h>
int g = 0;
void *myThreadFun(void *vargp)
{
    int myid = (int)vargp;
    static int s = 0;
    ++s;
    ++g;

    printf("Thread ID: %d, Static: %d, Global: %d\n", myid, ++s, ++g);
}

int main()
{
    int i;
    pthread_t tid;

    for (i = 0; i < 3; i++)
        pthread_create(&tid, NULL, myThreadFun, (void *)i);

    pthread_exit(NULL);
    return 0;
}
```

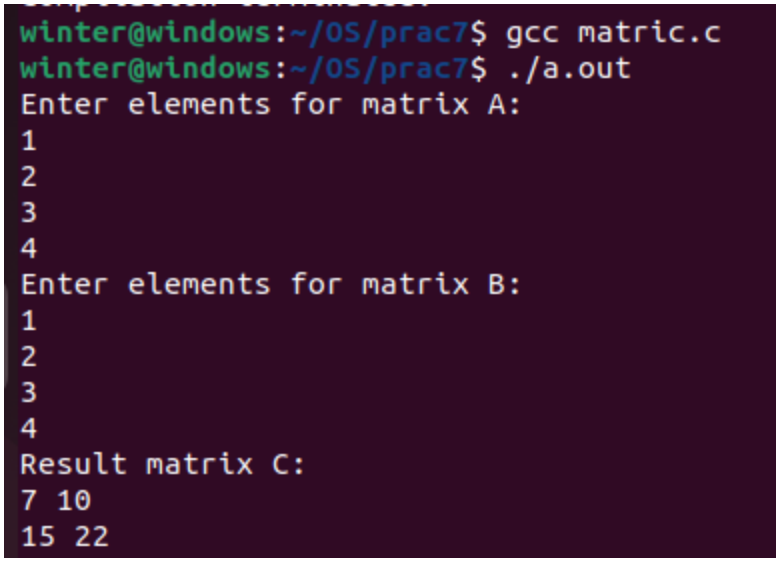
```
winter@windows:~/OS/prac7$ gedit p2.c
^C
winter@windows:~/OS/prac7$ gcc p2.c
p2.c: In function ‘myThreadFun’:
p2.c:6:20: warning: cast from pointer to integer of different size [-Wpointer-
o-int-cast]
    6 |         int myid = (int)vargp;
      |                    ^
p2.c: In function ‘main’:
p2.c:16:49: warning: cast to pointer from integer of different size [-Wint-to-
ointer-cast]
   16 |         pthread_create(&tid, NULL, myThreadFun, (void *)i);
      |                                                 ^
winter@windows:~/OS/prac7$ ./a.out
Thread ID: 1, Static: 2, Global: 2
Thread ID: 0, Static: 4, Global: 4
Thread ID: 2, Static: 6, Global: 6
winter@windows:~/OS/prac7$
```

Program 3- Matrix Multiplication using Threads

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

#define MAX_THREADS 8

// Structure to hold matrix information
typedef struct {
    int** matrixA;
    int** matrixB;
    int** result;
} MatrixData;

// Function to perform matrix multiplication for a given range of rows
void* multiplyRows(void* arg) {
    MatrixData* data = (MatrixData*)arg;

    int** matrixA = data->matrixA;
    int** matrixB = data->matrixB;
    int** result = data->result;

    // Perform matrix multiplication
    result[0][0] = matrixA[0][0] * matrixB[0][0] + matrixA[0][1] * matrixB[1][0];
    result[0][1] = matrixA[0][0] * matrixB[0][1] + matrixA[0][1] * matrixB[1][1];
    result[1][0] = matrixA[1][0] * matrixB[0][0] + matrixA[1][1] * matrixB[1][0];
```

```
    result[1][1] = matrixA[1][0] * matrixB[0][1] + matrixA[1][1] * matrixB[1][1];

    pthread_exit(NULL);
}

// Function to create threads and perform matrix multiplication
void multiplyMatrices(int** matrixA, int** matrixB, int** result) {
    pthread_t threads[MAX_THREADS];
    MatrixData data[MAX_THREADS];
    int threadCount = 1;  // We only need one thread for a 2x2 matrix multiplication

    for (int i = 0; i < threadCount; i++) {
        data[i].matrixA = matrixA;
        data[i].matrixB = matrixB;
        data[i].result = result;

        if (pthread_create(&threads[i], NULL, multiplyRows, (void*)&data[i]) != 0) {
            fprintf(stderr, "Failed to create thread %d\n", i);
            exit(1);
        }
    }

    // Wait for the thread to finish
    pthread_join(threads[0], NULL);
}

// Function to allocate memory for a matrix
int** allocateMatrix(int rows, int cols) {
    int** matrix = (int**)malloc(rows * sizeof(int*));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
    }
    return matrix;
}

// Function to free memory allocated for a matrix
void freeMatrix(int** matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

// Function to print a matrix
```

```
void printMatrix(int** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    // Allocate memory for matrices A, B, and C
    int** matrixA = allocateMatrix(2, 2);
    int** matrixB = allocateMatrix(2, 2);
    int** result = allocateMatrix(2, 2);

    // Get inputs for matrices A and B
    printf("Enter elements for matrix A:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            scanf("%d", &matrixA[i][j]);
        }
    }

    printf("Enter elements for matrix B:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            scanf("%d", &matrixB[i][j]);
        }
    }

    // Perform matrix multiplication using threads
    multiplyMatrices(matrixA, matrixB, result);

    // Print the result matrix
    printf("Result matrix C:\n");
    printMatrix(result, 2, 2);

    // Free memory for matrices
    freeMatrix(matrixA, 2);
    freeMatrix(matrixB, 2);
    freeMatrix(result, 2);

    return 0;
}
```

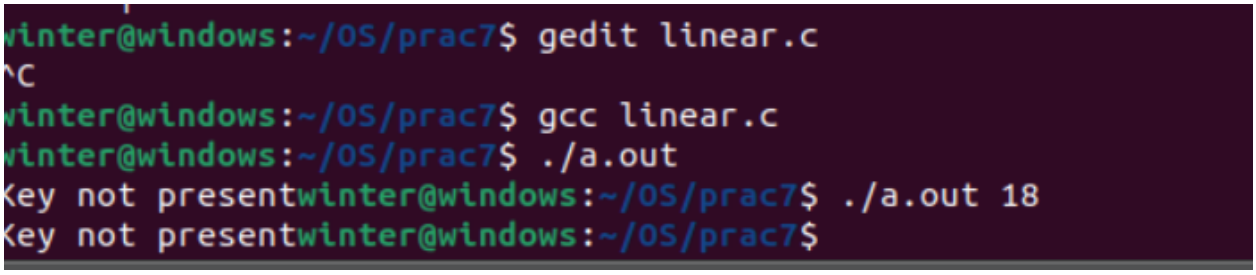


Program 4- Linear search using Multi-threading

```
// C code to search for element in a

// very large file using Multithreading

#include <.....>

#include <pthread.h>


// Max size of array

#define max 16


// Max number of threads to create

#define thread_max 4


int a[max] = { 1, 5, 7, 10, 12, 14, 15,

            18, 20, 22, 25, 27, 30,
```

```
            64, 110, 220 };
int key = 202;

// Flag to indicate if key is found in a[]
// or not.
int f = 0;

intcurrent_thread = 0;

// Linear search function which will
// run for all the threads
void* ThreadSearch(void* args)
{
    intnum = current_thread++; int i;

    for(i = num * (max / 4); i < ((num + 1) * (max / 4)); i++)
    {
        if(a[i] == key)
            f = 1;
    }
}

// Driver Code
```

```
int main()
{
   pthread_t thread[thread_max];
int i;

   for(i = 0; i <thread_max; i++)
         {
      pthread_create(&thread[i], NULL,
                  ThreadSearch, (void*)NULL);
   }

   for(i = 0; i <thread_max; i++)
         {
      pthread_join(thread[i], NULL);
   }

   if(f == 1)
      printf( "Key element found");
   else
      printf("Key not present");
   return0;
}
```

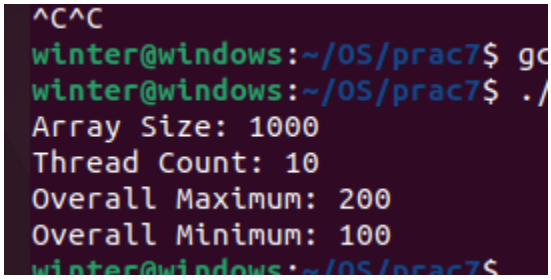


Program 5- Maximum and minimum element in an array

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

#define ARRAY_SIZE 1000
#define THREAD_COUNT 10

typedef struct {
    int* array;
    int start;
    int end;
    int max;
    int min;
} ThreadData;

void* findMaxMin(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    int* array = data->array;
    int start = data->start;
    int end = data->end;

    // Initialize max and min with the first element of the segment
    data->max = array[start];
    data->min = array[start];

    // Find max and min within the segment
    for (int i = start + 1; i <= end; i++) {
        if (array[i] > data->max) {
            data->max = array[i];
        }
```

```
        if (array[i] < data->min) {
            data->min = array[i];
        }
    }

    pthread_exit(NULL);
}

int main() {
    int array[ARRAY_SIZE];

    // Fill the array with random numbers between 100 and 200
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = rand() % 101 + 100;
    }

    // Create threads
    pthread_t threads[THREAD_COUNT];
    ThreadData threadData[THREAD_COUNT];
    int segmentSize = ARRAY_SIZE / THREAD_COUNT;

    for (int i = 0; i < THREAD_COUNT; i++) {
        threadData[i].array = array;
        threadData[i].start = i * segmentSize;
        threadData[i].end = (i + 1) * segmentSize - 1;

        if (pthread_create(&threads[i], NULL, findMaxMin, (void*)&threadData[i]) != 0) {
            fprintf(stderr, "Failed to create thread %d\n", i);
            exit(1);
        }
    }

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

    // Find overall maximum and minimum values
    int overallMax = threadData[0].max;
    int overallMin = threadData[0].min;
```

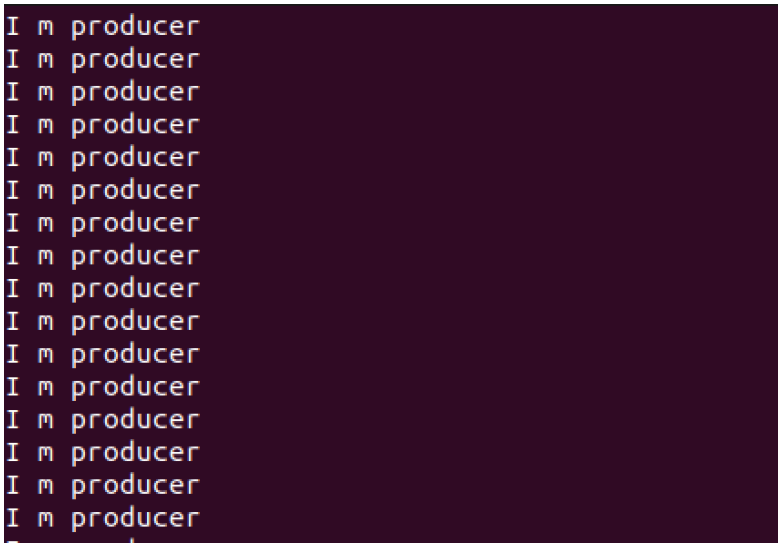
```
    for (int i = 1; i < THREAD_COUNT; i++) {
        if (threadData[i].max > overallMax) {
            overallMax = threadData[i].max;
        }
        if (threadData[i].min < overallMin) {
            overallMin = threadData[i].min;
        }
    }

    // Print the results
    printf("Array Size: %d\n", ARRAY_SIZE);
    printf("Thread Count: %d\n", THREAD_COUNT);
    printf("Overall Maximum: %d\n", overallMax);
    printf("Overall Minimum: %d\n", overallMin);

    return 0;
}
```



Program 6- Producer Consumer without synchronization

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

void *producer(); /* the thread */
void *consumer(); /* the thread */

int main() {

pthread_t ptid, ctid;       //Thread ID
pthread_create(&ptid,NULL,producer,NULL);
pthread_create(&ctid,NULL,consumer,NULL);
pthread_join(ptid, NULL);
pthread_join(ctid, NULL);
```

```
    }

//The thread will begin control in this function
void *producer(void *param) {
do{
printf("I m producer\n");

}while(1);
pthread_exit(0);
}

//The thread will begin control in this function
void *consumer(void *param) {
do{
printf("I m consumer\n");
}while(1);
pthread_exit(0);
}
```



Program 7- Producer Consumer with synchronization

```
# include <stdio.h>
# include <stdlib.h>
# include <pthread.h>
# define BufferSize 10

void *Producer();
void *Consumer();
```

```
int BufferIndex= -1;
char BUFFER[10];

pthread_cond_t  Buffer_Empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t  Buffer_Full =PTHREAD_COND_INITIALIZER;
pthread_mutex_t mVar=PTHREAD_MUTEX_INITIALIZER;

int main()
{
pthread_t ptid,ctid;


pthread_create(&ptid,NULL,Producer,NULL);
pthread_create(&ctid,NULL,Consumer,NULL);

pthread_join(ptid,NULL);
pthread_join(ctid,NULL);

return 0;
}

void *Producer()
{

   //do
int i;
for(i=0; i<15 ; i++)
   {
pthread_mutex_lock(&mVar);
if(BufferIndex==BufferSize-1)
pthread_cond_wait(&Buffer_Empty,&mVar);

BUFFER[++BufferIndex]='#';
printf("Produce : %d \n",BufferIndex);
pthread_mutex_unlock(&mVar);
pthread_cond_signal(&Buffer_Full);

   }//while(1);

}

void *Consumer()
{

   //do
int i;
```

```
for(i=0; i<15 ; i++)
   {
pthread_mutex_lock(&mVar);
if(BufferIndex==-1) {
pthread_cond_wait(&Buffer_Full,&mVar);
}
printf("Consume : %d \n",BufferIndex--);
pthread_mutex_unlock(&mVar);
pthread_cond_signal(&Buffer_Empty);

   }//while(1);
}
```

```
winter@windows:~/OS/prac7$ gcc SyncPC.c -lpthread
winter@windows:~/OS/prac7$ ./a.out
Produce : 0
Produce : 1
Produce : 2
Produce : 3
Produce : 4
Produce : 5
Produce : 6
Produce : 7
Produce : 8
Produce : 9
Consume : 9
Consume : 8
Consume : 7
Consume : 6
Consume : 5
Consume : 4
```

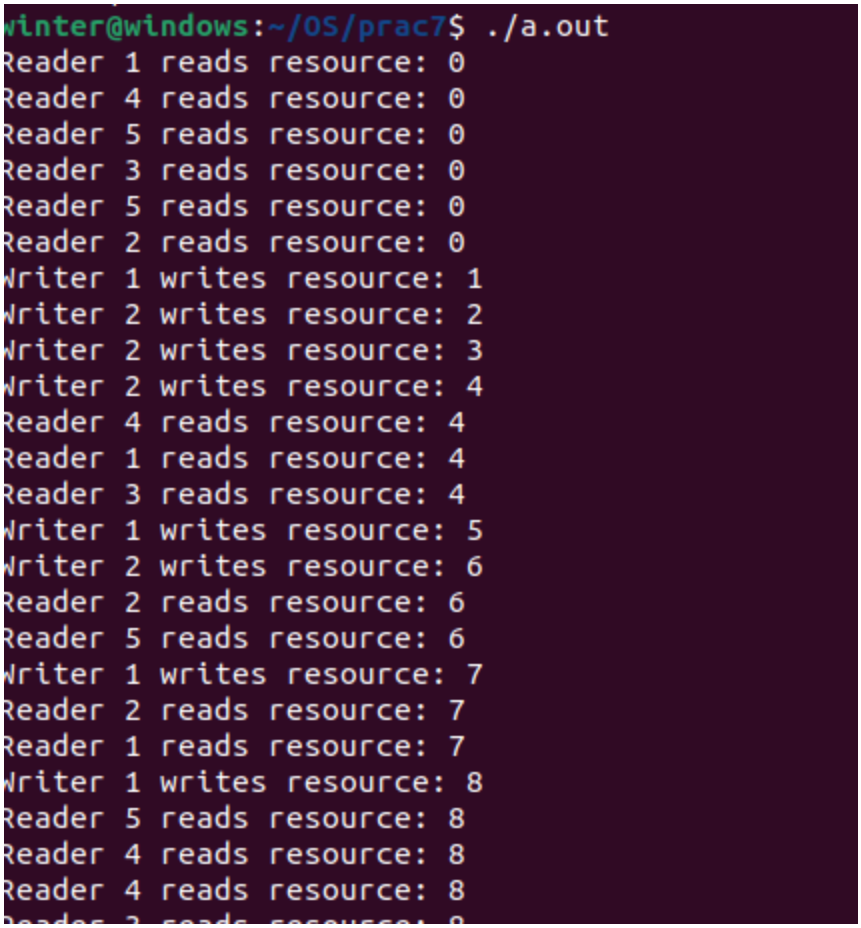
Program 8- Readers Writers with mutex and pthread.

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

#define READERS_COUNT 5
```

```
#define WRITERS_COUNT 2

pthread_mutex_t mutex;
pthread_cond_t readCondition, writeCondition;
int readers = 0;
int resource = 0;

void* reader(void* arg) {
    int readerId = *(int*)arg;
    while (1) {
        pthread_mutex_lock(&mutex);

        // Wait until there are no writers or writing is in progress
        while (readers == -1) {
            pthread_cond_wait(&readCondition, &mutex);
        }

        readers++;
        pthread_mutex_unlock(&mutex);

        // Read the shared resource
        printf("Reader %d reads resource: %d\n", readerId, resource);

        pthread_mutex_lock(&mutex);
        readers--;

        // Signal the waiting writers if no readers are left
        if (readers == 0) {
            pthread_cond_signal(&writeCondition);
        }

        pthread_mutex_unlock(&mutex);

        // Sleep for a random period of time
        sleep(rand() % 3);
    }

    pthread_exit(NULL);
}
```

```
void* writer(void* arg) {
    int writerId = *(int*)arg;
    while (1) {
        pthread_mutex_lock(&mutex);

        // Wait until there are no readers or writers
        while (readers != 0) {
            pthread_cond_wait(&writeCondition, &mutex);
        }

        readers = -1;  // Indicate writing is in progress

        pthread_mutex_unlock(&mutex);

        // Write to the shared resource
        resource++;
        printf("Writer %d writes resource: %d\n", writerId, resource);

        pthread_mutex_lock(&mutex);
        readers = 0;

        // Signal the waiting readers and writers
        pthread_cond_broadcast(&readCondition);
        pthread_cond_signal(&writeCondition);

        pthread_mutex_unlock(&mutex);

        // Sleep for a random period of time
        sleep(rand() % 3);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t readerThreads[READERS_COUNT];
    pthread_t writerThreads[WRITERS_COUNT];
    int readerIds[READERS_COUNT];
    int writerIds[WRITERS_COUNT];
```

```
    // Initialize mutex and conditions
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&readCondition, NULL);
    pthread_cond_init(&writeCondition, NULL);

    // Create reader threads
    for (int i = 0; i < READERS_COUNT; i++) {
        readerIds[i] = i + 1;
        pthread_create(&readerThreads[i], NULL, reader, &readerIds[i]);
    }

    // Create writer threads
    for (int i = 0; i < WRITERS_COUNT; i++) {
        writerIds[i] = i + 1;
        pthread_create(&writerThreads[i], NULL, writer, &writerIds[i]);
    }

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

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

    // Destroy mutex and conditions
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&readCondition);
    pthread_cond_destroy(&writeCondition);

    return 0;
}
```



RESULT -

Linux C programs to demonstrate the concept of threads and semaphores for process synchronization have been implemented.