Title: Implement Multithread Model

In this C program, we will implement a simple multithreading model using the `pthread` library, which is widely used for creating and managing threads in C. The program will demonstrate the basic concepts of multithreading, such as creating threads, synchronizing them, and sharing data between threads.

Key Concepts

1. **Thread Creation**: We will create multiple threads using `pthread_create()`.
2. **Thread Synchronization**: To synchronize threads, we can use a mutex (mutual exclusion) to ensure that shared resources are accessed safely.
3. **Joining Threads**: After creating threads, we need to wait for them to finish their execution using `pthread_join()`.

Program Explanation

The program will have the following structure:

- **Thread function**: A simple function that performs some task (e.g., printing a message).
- **Main function**: The main function will create multiple threads, start them, and then wait for them to finish.

C Program

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

// Number of threads to create
#define NUM_THREADS 5

// Shared resource (to demonstrate thread synchronization)
int counter = 0;
pthread_mutex_t counter_mutex;  // Mutex to protect shared resource

// Thread function that increments the counter
void *increment_counter(void *thread_id) {
    long tid = (long)thread_id;

    // Lock the mutex before accessing the shared resource
    pthread_mutex_lock(&counter_mutex);

    printf("Thread %ld: Incrementing counter...\n", tid);
    counter++;  // Critical section: increment the counter
    printf("Thread %ld: Counter value is now %d\n", tid, counter);

    // Unlock the mutex after accessing the shared resource
    pthread_mutex_unlock(&counter_mutex);
```

```
    pthread_exit(NULL);  // Exit the thread
}

int main() {
    pthread_t threads[NUM_THREADS];  // Array to store thread identifiers
    long t;

    // Initialize the mutex
    if (pthread_mutex_init(&counter_mutex, NULL) != 0) {
        printf("Mutex initialization failed!\n");
        return 1;
    }

    // Create threads
    for (t = 0; t < NUM_THREADS; t++) {
        printf("Main: Creating thread %ld\n", t);
        int ret = pthread_create(&threads[t], NULL, increment_counter, (void
*)t);
        if (ret) {
            printf("Error creating thread %ld: %d\n", t, ret);
            exit(-1);
        }
    }

    // Wait for all threads to complete
    for (t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
        printf("Main: Thread %ld has finished.\n", t);
    }

    // Destroy the mutex
    pthread_mutex_destroy(&counter_mutex);

    // Print the final value of the counter
    printf("Main: Final counter value is %d\n", counter);

    return 0;
}
```

Explanation of the Code

1. **Includes**:
   - `stdio.h`: For input and output operations.
   - `stdlib.h`: For standard library functions.
   - `pthread.h`: For working with threads in C.
2. **Global Variables**:
   - `counter`: A shared resource that will be accessed by multiple threads.
   - `counter_mutex`: A mutex (mutual exclusion lock) used to protect the shared `counter` variable.
3. **`increment_counter` Function**:

    - This function is called by each thread. It locks the mutex, increments the shared `counter`, prints the updated counter value, and then unlocks the mutex. The critical section is protected by the mutex to prevent race conditions.
4. **Main Function**:
    - We first initialize the mutex using `pthread_mutex_init()`.
    - A loop is used to create `NUM_THREADS` threads. Each thread runs the `increment_counter` function, passing its thread index (`t`) as an argument.
    - After creating the threads, the main function waits for all threads to complete using `pthread_join()`.
    - Finally, the mutex is destroyed using `pthread_mutex_destroy()`.
5. **Mutex Locking and Unlocking**:
    - `pthread_mutex_lock()` locks the mutex before accessing the shared resource (`counter`), ensuring only one thread can modify the counter at a time.
    - `pthread_mutex_unlock()` unlocks the mutex, allowing other threads to access the shared resource.
6. **Thread Exit**:
    - `pthread_exit(NULL)` is called at the end of the thread function to ensure the thread finishes its execution properly.

Key Functions

- **`pthread_create`**: Creates a new thread that starts executing the specified function.
- **`pthread_join`**: Makes the calling thread (main thread) wait for the specified thread to finish.
- **`pthread_mutex_lock`**: Locks the mutex, preventing other threads from accessing the shared resource.
- **`pthread_mutex_unlock`**: Unlocks the mutex, allowing other threads to access the shared resource.
- **`pthread_mutex_init`**: Initializes the mutex before use.
- **`pthread_mutex_destroy`**: Destroys the mutex after it is no longer needed.

Explanation of Output:

- The threads run concurrently, each incrementing the `counter` variable.
- Due to the mutex locking and unlocking mechanism, only one thread can increment the counter at a time, ensuring thread safety.
- The final value of `counter` is `5`, as expected, since each of the 5 threads incremented it once.

Concept of Semaphores:

- **P (Proberen)** operation: It decrements the semaphore. If the value is greater than 0, it allows the process to proceed. If the value is 0, the process is blocked.
- **V (Verhogen)** operation: It increments the semaphore, potentially waking up a blocked process.

Key Functions:

- `sem_init()`: Initializes a semaphore.
- `sem_wait()`: Performs the **P** operation (decrements the semaphore and potentially blocks the process).
- `sem_post()`: Performs the **V** operation (increments the semaphore and potentially wakes up a blocked process).
- `sem_destroy()`: Destroys the semaphore.

C Program to Demonstrate Semaphores

In this example, we'll create a simple producer-consumer problem using semaphores. A producer thread will produce items and a consumer thread will consume those items. Semaphores will be used to synchronize access to the shared resource.

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>


#define BUFFER_SIZE 5

#define MAX_ITEMS 20  // Set a limit for the total number of items to be produced and consumed


// Shared buffer (a simple circular buffer)

int buffer[BUFFER_SIZE];

int in = 0; // Index for the next item to produce

int out = 0; // Index for the next item to consume


// Semaphores
```

```
sem_t empty;   // Semaphore to track empty spaces in the buffer
sem_t full;    // Semaphore to track full spaces in the buffer
sem_t mutex;   // Semaphore to protect the critical section (buffer access)

// Shared variables to track the number of items produced and consumed
int produced_items = 0;
int consumed_items = 0;

// Producer thread function
void* producer(void* arg) {
   int item;
   while (produced_items < MAX_ITEMS) {  // Run until the max number of items is produced
      item = rand() % 100;  // Generate a random item to produce

      // Wait for an empty slot in the buffer
      sem_wait(&empty);

      // Enter critical section (protect the buffer)
      sem_wait(&mutex);

      // Produce the item and place it in the buffer
      buffer[in] = item;
      printf("Producer produced: %d at index %d\n", item, in);
      in = (in + 1) % BUFFER_SIZE;  // Move to the next slot

      // Leave the critical section
      sem_post(&mutex);

      // Signal that there's a new full slot in the buffer
```

```
        sem_post(&full);

        // Increment the produced item count
        produced_items++;

        // Sleep to simulate production time
        sleep(1);
    }
    pthread_exit(NULL);  // Exit the producer thread
}

// Consumer thread function
void* consumer(void* arg) {
    int item;
    while (consumed_items < MAX_ITEMS) {  // Run until the max number of items is consumed
        // Wait for a full slot in the buffer
        sem_wait(&full);

        // Enter critical section (protect the buffer)
        sem_wait(&mutex);

        // Consume the item from the buffer
        item = buffer[out];
        printf("Consumer consumed: %d from index %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;  // Move to the next slot

        // Leave the critical section
        sem_post(&mutex);
```

```
        // Signal that there's a new empty slot in the buffer
        sem_post(&empty);

        // Increment the consumed item count
        consumed_items++;

        // Sleep to simulate consumption time
        sleep(2);
    }
    pthread_exit(NULL);  // Exit the consumer thread
}

int main() {
    pthread_t producer_thread, consumer_thread;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);  // Initially, all slots are empty
    sem_init(&full, 0, 0);             // Initially, no slots are full
    sem_init(&mutex, 0, 1);            // Mutex is initially available (1)

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Clean up semaphores
```

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

    // Final message after execution limit is reached
    printf("Execution limit reached. Producer produced %d items and Consumer consumed %d items.\n", produced_items, consumed_items);

    return 0;
}
```

Explanation of the Code:

1. **Shared Variables**:
   - `buffer[]`: This is the circular buffer where items are stored and retrieved. The buffer size is defined by `BUFFER_SIZE`.
   - `in` and `out`: These are the indices for the next item to produce and the next item to consume, respectively.
2. **Semaphores**:
   - `empty`: A counting semaphore initialized to the size of the buffer (`BUFFER_SIZE`). It tracks the number of empty slots in the buffer.
   - `full`: A counting semaphore initialized to 0. It tracks the number of full slots in the buffer.
   - `mutex`: A binary semaphore initialized to 1. It is used to ensure mutual exclusion when accessing the shared buffer, ensuring that only one thread can modify the buffer at a time.
3. **Producer Thread (`producer` function)**:
   - The producer generates random items and tries to place them in the buffer.
   - The producer waits for an empty slot in the buffer using `sem_wait(&empty)`.
   - Once an empty slot is available, the producer locks the `mutex` to ensure exclusive access to the shared buffer.
   - After producing an item and placing it in the buffer, the producer releases the `mutex` and signals that a new full slot is available by calling `sem_post(&full)`.
4. **Consumer Thread (`consumer` function)**:
   - The consumer waits for a full slot in the buffer using `sem_wait(&full)`.
   - Once a full slot is available, the consumer locks the `mutex` to ensure exclusive access to the buffer.
   - After consuming an item, the consumer releases the `mutex` and signals that a new empty slot is available by calling `sem_post(&empty)`.

5. **Main Function**:
    - The main function initializes the semaphores (`empty`, `full`, and `mutex`) and creates two threads: one for the producer and one for the consumer.
    - The main thread waits for both threads to complete using `pthread_join()`. (In this example, the threads run indefinitely.)
    - The semaphores are destroyed at the end to clean up the resources.

Key Concepts:

- **Semaphores** are used to ensure synchronization between the producer and consumer, avoiding race conditions.
- **Counting semaphores** (`empty` and `full`) are used to track available and occupied slots in the buffer.
- **Binary semaphore** (`mutex`) is used to ensure mutual exclusion when accessing the shared buffer.