Exam 3 Study Guide

Concurrency in C: Threads, Mutexes, Semaphores, Read-Write Locks, and Barriers

Table of Contents

POSIX Threads (Pthreads) Overview

Pthreads (POSIX Threads) is the most common threading library in Unix-based systems. The API provides functions to create, manage, and synchronize threads.

Basic APIs for Thread Management:

- `pthread_create()`: Creates a new thread.
    - **Syntax:**

    ```
    int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void *), void *arg);
    ```

        - **thread**: Pointer to `pthread_t` where the thread ID will be stored.
        - **attr**: Attributes for the new thread (can be `NULL` for default attributes).
        - **start_routine**: Function that the thread will execute.
        - **arg**: Argument to pass to `start_routine`.
        - **Returns**: `0` on success, an error number on failure.
        - **Example:**

    ```
    pthread_t thread;
    int ret = pthread_create(&thread, NULL, thread_function, NULL);
    ```

```c
if (ret != 0) {
    printf("Error creating thread\n");
}
```

- `pthread_join()`: Waits for a thread to finish.
  - **Syntax:**

    ```c
    int pthread_join(pthread_t thread, void **retval);
    ```

    - **thread**: The thread ID to wait for.
    - **retval**: Pointer to the return value of the thread (can be `NULL`).
    - **Returns**: `0` on success, an error number on failure.
    - **Example:**

    ```c
    pthread_join(thread, NULL); // Wait for the specified thread to finish
    ```

- `pthread_exit()`: Terminates the calling thread.
  - **Syntax:**

    ```c
    void pthread_exit(void *retval);
    ```

    - **retval**: Pointer to the return value for the calling thread.
    - **Usage**: Used by a thread to exit and optionally return a value.
    - **Example:**

    ```c
    pthread_exit(NULL); // Exit the current thread
    ```

- `pthread_detach()`: Detaches a thread, allowing its resources to be released when it terminates without requiring a join.
  - **Syntax:**

    ```c
    int pthread_detach(pthread_t thread);
    ```

    - **thread**: The thread ID to detach.
    - **Returns**: `0` on success, an error number on failure.
    - **Example:**

    ```c
    pthread_detach(thread); // Detach the specified thread
    ```

Example Usage:

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

void* thread_function(void* arg) {
    int id = *((int*)arg);
    printf("Hello from thread %d!\n", id);
    free(arg); // Free dynamically allocated memory
    pthread_exit(NULL); // Exit thread
}

int main() {
    pthread_t threads[5];
    for (int i = 0; i < 5; i++) {
        int* id = malloc(sizeof(int));
        *id = i;
        pthread_create(&threads[i], NULL, thread_function, id); // Create
thread
    }
    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL); // Wait for each thread to finish
    }
    return 0;
}
```

In the above example, we dynamically allocate memory for thread arguments to avoid data races on the loop variable `i`. Each thread prints its unique ID.

Detached Threads

Detached threads automatically release their resources upon completion without requiring a `pthread_join()` call.

Example Usage:

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

void* thread_function(void* arg) {
    printf("Detached thread running.\n");
    pthread_exit(NULL); // Exit the current thread
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_function, NULL); // Create thread
    pthread_detach(thread); // Detach thread
    // The main thread continues executing without waiting for the detached
```

```
thread.
    printf("Main thread done.\n");
    return 0;
}
```

In this example, the detached thread runs independently of the main thread, which avoids having to call `pthread_join()`.

Mutexes

Introduction to Mutexes

A mutex (mutual exclusion) is a synchronization primitive that ensures only one thread can access a critical section at a time, preventing data races.

Building Mutexes from Scratch

The simplest way to conceptualize a mutex is as a "lock" that only one thread can hold. In the kernel, mutexes can be implemented using atomic variables and system calls that put threads to sleep when the lock is not available.

Basic APIs for Mutex Management:

- `pthread_mutex_init()`: Initializes a mutex.
  - **Syntax:**

    ```
    int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *attr);
    ```

    - **mutex**: Pointer to the mutex to initialize.
    - **attr**: Attributes for the mutex (can be `NULL` for default attributes).
    - **Returns**: `0` on success, an error number on failure.
    - **Example:**

    ```
    pthread_mutex_t lock;
    pthread_mutex_init(&lock, NULL); // Initialize the mutex
    ```

- `pthread_mutex_lock()`: Locks the mutex.
  - **Syntax:**

    ```
    int pthread_mutex_lock(pthread_mutex_t *mutex);
    ```

    - **mutex**: The mutex to lock.
    - **Returns**: `0` on success, an error number on failure.
    - **Example:**

```
pthread_mutex_lock(&lock); // Lock the mutex
```

- `pthread_mutex_unlock()` : Unlocks the mutex.
  - **Syntax:**

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - **mutex**: The mutex to unlock.
  - **Returns**: `0` on success, an error number on failure.
  - **Example:**

```
pthread_mutex_unlock(&lock); // Unlock the mutex
```

- `pthread_mutex_destroy()` : Destroys the mutex.
  - **Syntax:**

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

  - **mutex**: The mutex to destroy.
  - **Returns**: `0` on success, an error number on failure.
  - **Example:**

```
pthread_mutex_destroy(&lock); // Destroy the mutex
```

Example Usage:

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

pthread_mutex_t lock;
int counter = 0;

void* increment(void* arg) {
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&lock); // Lock the mutex
        counter++;
        printf("Thread %ld incremented counter to %d\n", pthread_self(),
counter);
        pthread_mutex_unlock(&lock); // Unlock the mutex
    }
    pthread_exit(NULL); // Exit the current thread
}

int main() {
    pthread_t thread1, thread2;
```

```
    pthread_mutex_init(&lock, NULL); // Initialize the mutex

    pthread_create(&thread1, NULL, increment, NULL); // Create thread 1
    pthread_create(&thread2, NULL, increment, NULL); // Create thread 2

    pthread_join(thread1, NULL); // Wait for thread 1
    pthread_join(thread2, NULL); // Wait for thread 2

    pthread_mutex_destroy(&lock); // Destroy the mutex
    return 0;
}
```

In the above example, we protect the `counter` variable using a mutex to prevent data races when multiple threads try to increment the value concurrently.

Deadlock Prevention

Deadlocks can occur when multiple threads are waiting indefinitely for resources held by each other. Deadlock prevention techniques include:

- **Lock Ordering**: Ensure all threads acquire locks in a predefined order.
- **Timeouts**: Use timed lock operations, such as `pthread_mutex_timedlock()`, to avoid indefinite blocking.

POSIX Conditions

POSIX conditions (or condition variables) allow threads to wait for certain conditions to be met. They are often used in conjunction with mutexes to manage shared state.

Basic APIs for Condition Variables:

- `pthread_cond_init()`: Initializes a condition variable.
  - **Syntax:**

    ```
    int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
    ```

    - **cond**: Pointer to the condition variable to initialize.
    - **attr**: Attributes for the condition variable (can be `NULL` for default attributes).
    - **Returns**: `0` on success, an error number on failure.
    - **Example:**

    ```
    pthread_cond_t cond;
    pthread_cond_init(&cond, NULL); // Initialize the condition variable
    ```

- `pthread_cond_wait()`: Waits for the condition (must be used with a locked mutex).

- **Syntax:**

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

  - **cond**: The condition variable to wait on.
  - **mutex**: The mutex associated with the condition.
  - **Returns**: `0` on success, an error number on failure.
  - **Example:**

```
pthread_mutex_lock(&lock); // Lock the mutex
pthread_cond_wait(&cond, &lock); // Wait for the condition
pthread_mutex_unlock(&lock); // Unlock the mutex
```

- `pthread_cond_signal()`: Wakes up one waiting thread.
  - **Syntax:**

```
int pthread_cond_signal(pthread_cond_t *cond);
```

  - **cond**: The condition variable to signal.
  - **Returns**: `0` on success, an error number on failure.
  - **Example:**

```
pthread_cond_signal(&cond); // Signal the condition
```

- `pthread_cond_broadcast()`: Wakes up all waiting threads.
  - **Syntax:**

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

  - **cond**: The condition variable to broadcast.
  - **Returns**: `0` on success, an error number on failure.
  - **Example:**

```
pthread_cond_broadcast(&cond); // Broadcast to all waiting threads
```

- `pthread_cond_destroy()`: Destroys the condition variable.
  - **Syntax:**

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

  - **cond**: The condition variable to destroy.
  - **Returns**: `0` on success, an error number on failure.
  - **Example:**

```
pthread_cond_destroy(&cond); // Destroy the condition variable
```

Example Usage:

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

pthread_mutex_t lock;
pthread_cond_t cond;
int ready = 0;

void* wait_for_condition(void* arg) {
    pthread_mutex_lock(&lock); // Lock the mutex
    while (!ready) {
        pthread_cond_wait(&cond, &lock); // Wait for the condition
    }
    printf("Thread %ld proceeding after condition met.\n", pthread_self());
    pthread_mutex_unlock(&lock); // Unlock the mutex
    pthread_exit(NULL); // Exit the current thread
}

void* signal_condition(void* arg) {
    pthread_mutex_lock(&lock); // Lock the mutex
    ready = 1;
    pthread_cond_signal(&cond); // Signal the condition
    pthread_mutex_unlock(&lock); // Unlock the mutex
    pthread_exit(NULL); // Exit the current thread
}

int main() {
    pthread_t thread1, thread2;
    pthread_mutex_init(&lock, NULL); // Initialize the mutex
    pthread_cond_init(&cond, NULL); // Initialize the condition variable

    pthread_create(&thread1, NULL, wait_for_condition, NULL); // Create thread 1
    pthread_create(&thread2, NULL, signal_condition, NULL); // Create thread 2

    pthread_join(thread1, NULL); // Wait for thread 1
    pthread_join(thread2, NULL); // Wait for thread 2

    pthread_mutex_destroy(&lock); // Destroy the mutex
    pthread_cond_destroy(&cond); // Destroy the condition variable
    return 0;
}
```

In this example, `thread1` waits until `ready` is set to 1, and `thread2` signals the condition to allow `thread1` to proceed.

Building Semaphores from First Principles

Semaphores are synchronization tools that can be built from basic synchronization primitives like mutexes and condition variables. In this section, we will implement a semaphore using POSIX mutexes and condition variables. This approach is often used when the standard semaphore library is unavailable or if more control is desired.

Using POSIX Conditions and Mutexes to Create a Semaphore

Below is an implementation of a semaphore from first principles using POSIX condition variables (`pthread_cond_t`) and mutexes (`pthread_mutex_t`).

APIs for Custom Semaphore Management:

- `pthread_sema_init()`: Initializes the semaphore.
  - **Syntax:**

    ```
    int pthread_sema_init(pthread_sema_t *sem, unsigned int value);
    ```

    - **sem**: Pointer to the semaphore structure.
    - **value**: Initial value of the semaphore counter.
    - **Returns**: `0` on success, `-1` on failure.
    - **Example:**

    ```
    pthread_sema_t sem;
    pthread_sema_init(&sem, 2); // Initialize the semaphore to allow up to 2 threads in the critical section
    ```

- `pthread_sema_destroy()`: Destroys the semaphore.
  - **Syntax:**

    ```
    int pthread_sema_destroy(pthread_sema_t *sem);
    ```

    - **sem**: Pointer to the semaphore structure.
    - **Returns**: `0` on success.
    - **Example:**

    ```
    pthread_sema_destroy(&sem); // Destroy the semaphore
    ```

- `pthread_sema_post()`: Signals (increments) the semaphore.
  - **Syntax:**

```c
int pthread_sema_post(pthread_sema_t *sem);
```

- **sem**: Pointer to the semaphore structure.
- **Returns**: `0` on success.
- **Example:**

```c
pthread_sema_post(&sem); // Signal the semaphore
```

- `pthread_sema_wait()`: Waits (decrements) the semaphore.
  - **Syntax:**

```c
int pthread_sema_wait(pthread_sema_t *sem);
```

- **sem**: Pointer to the semaphore structure.
- **Returns**: `0` on success.
- **Example:**

```c
pthread_sema_wait(&sem); // Wait for the semaphore
```

- `pthread_sema_trywait()`: Attempts to wait for the semaphore without blocking.
  - **Syntax:**

```c
int pthread_sema_trywait(pthread_sema_t *sem);
```

- **sem**: Pointer to the semaphore structure.
- **Returns**: `0` on success, `EAGAIN` if the semaphore is not available.
- **Example:**

```c
if (pthread_sema_trywait(&sem) == EAGAIN) {
    printf("Semaphore not available\n");
}
```

Example Implementation and Usage:

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

typedef struct pthread_sema {
    pthread_mutex_t _m;
    pthread_cond_t _c;
    int _counter;
    int _asleep;
} pthread_sema_t;
```

```c
int pthread_sema_init(pthread_sema_t* sem, unsigned int v) {
    if (pthread_mutex_init(&sem->_m, NULL) != 0) return -1;
    if (pthread_cond_init(&sem->_c, NULL) != 0) {
        pthread_mutex_destroy(&sem->_m);
        return -1;
    }
    sem->_counter = v;
    sem->_asleep = 0;
    return 0;
}

int pthread_sema_destroy(pthread_sema_t* sem) {
    pthread_mutex_destroy(&sem->_m);
    pthread_cond_destroy(&sem->_c);
    return 0;
}

int pthread_sema_post(pthread_sema_t* sem) {
    pthread_mutex_lock(&sem->_m); // Lock the mutex
    sem->_counter++;
    if (sem->_asleep > 0) {
        pthread_cond_signal(&sem->_c); // Signal a waiting thread
    }
    pthread_mutex_unlock(&sem->_m); // Unlock the mutex
    return 0;
}

int pthread_sema_wait(pthread_sema_t* sem) {
    pthread_mutex_lock(&sem->_m); // Lock the mutex
    while (sem->_counter <= 0) {
        sem->_asleep++;
        pthread_cond_wait(&sem->_c, &sem->_m); // Wait for the condition
        sem->_asleep--;
    }
    sem->_counter--;
    pthread_mutex_unlock(&sem->_m); // Unlock the mutex
    return 0;
}

int pthread_sema_trywait(pthread_sema_t* sem) {
    int result = 0;
    pthread_mutex_lock(&sem->_m); // Lock the mutex
    if (sem->_counter > 0) {
        sem->_counter--;
    } else {
        result = EAGAIN;
    }
    pthread_mutex_unlock(&sem->_m); // Unlock the mutex
    return result;
```

```c
}

void* task(void* arg) {
    pthread_sema_t* sem = (pthread_sema_t*)arg;
    pthread_sema_wait(sem); // Wait (decrement) the semaphore
    printf("Thread %ld is in critical section.\n", pthread_self());
    pthread_sema_post(sem); // Signal (increment) the semaphore
    pthread_exit(NULL); // Exit the current thread
}

int main() {
    pthread_t threads[4];
    pthread_sema_t sem;

    pthread_sema_init(&sem, 2); // Initialize the semaphore to allow up to 2
threads in the critical section

    for (int i = 0; i < 4; i++) {
        pthread_create(&threads[i], NULL, task, &sem); // Create threads
    }

    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL); // Wait for each thread
    }

    pthread_sema_destroy(&sem); // Destroy the semaphore
    return 0;
}
```

In this implementation, a custom semaphore (`pthread_sema_t`) is created using a mutex and a condition variable. The semaphore can manage access to a critical section, allowing only a limited number of threads (in this case, 2) to enter at once.

UNIX Semaphores for Inter-Process Communication

UNIX System V semaphores are useful for inter-process communication (IPC). These semaphores are part of the System V IPC mechanisms and can be used by multiple processes, but require shared memory to coordinate between processes.

Key Concepts for UNIX Semaphores

- **Shared Memory Requirement**: UNIX semaphores are often used in conjunction with shared memory, allowing different processes to synchronize access to the shared resource.
- **Poor-Man's Threading**: When the `pthread` library is not available, UNIX semaphores can offer basic synchronization mechanisms akin to threading. This allows for simple coordination between processes without full-fledged thread support.

Basic APIs for UNIX Semaphores

- `semget()`: Creates a new semaphore or accesses an existing one.
  - **Syntax:**

    ```c
    int semget(key_t key, int nsems, int semflg);
    ```

    - **key**: Unique identifier for the semaphore set.
    - **nsems**: Number of semaphores in the set.
    - **semflg**: Flags to control creation and permissions.
    - **Returns**: Semaphore ID on success, `-1` on failure.
    - **Example:**

    ```c
    int sem_id = semget(IPC_PRIVATE, 1, IPC_CREAT | 0666);
    if (sem_id == -1) {
        perror("semget failed");
    }
    ```

- `semop()`: Performs operations on a semaphore.
  - **Syntax:**

    ```c
    int semop(int semid, struct sembuf *sops, size_t nsops);
    ```

    - **semid**: Semaphore ID returned by `semget()`.
    - **sops**: Pointer to an array of semaphore operations.
    - **nsops**: Number of operations to be performed.
    - **Returns**: `0` on success, `-1` on failure.
    - **Example:**

    ```c
    struct sembuf op;
    op.sem_num = 0;
    op.sem_op = -1; // Wait operation
    op.sem_flg = 0;
    if (semop(sem_id, &op, 1) == -1) {
        perror("semop failed");
    }
    ```

- `semctl()`: Controls semaphore properties.
  - **Syntax:**

    ```c
    int semctl(int semid, int semnum, int cmd, ...);
    ```

    - **semid**: Semaphore ID.
    - **semnum**: The index of the semaphore in the set.

- **cmd**: Command to perform (e.g., `IPC_RMID` to remove the semaphore).
- **Returns**: Depends on the command, `-1` on failure.
- **Example:**

```
if (semctl(sem_id, 0, IPC_RMID) == -1) {
    perror("semctl failed");
}
```

While UNIX semaphores provide a powerful way to manage resources between processes, they are typically more complex to use than POSIX semaphores.

Read-Write Locks

Introduction to Read-Write Locks

Read-Write locks allow multiple threads to read simultaneously while ensuring only one thread writes at a time. This provides better performance for read-heavy workloads.

Basic APIs for Read-Write Lock Management:

- `pthread_rwlock_init()`: Initializes a read-write lock.
  - **Syntax:**

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const
pthread_rwlockattr_t *attr);
```

- **rwlock**: Pointer to the read-write lock.
- **attr**: Attributes for the lock (can be `NULL` for default attributes).
- **Returns**: `0` on success, an error number on failure.
- **Example:**

```
pthread_rwlock_t lock;
pthread_rwlock_init(&lock, NULL); // Initialize the read-write lock
```

- `pthread_rwlock_rdlock()`: Locks for reading.
  - **Syntax:**

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

- **rwlock**: The read-write lock to lock for reading.
- **Returns**: `0` on success, an error number on failure.
- **Example:**

```
pthread_rwlock_rdlock(&lock); // Lock for reading
```

- `pthread_rwlock_wrlock()`: Locks for writing.
  - **Syntax:**

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

  - **rwlock**: The read-write lock to lock for writing.
  - **Returns**: `0` on success, an error number on failure.
  - **Example:**

```
pthread_rwlock_wrlock(&lock); // Lock for writing
```

- `pthread_rwlock_unlock()`: Unlocks the lock.
  - **Syntax:**

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

  - **rwlock**: The read-write lock to unlock.
  - **Returns**: `0` on success, an error number on failure.
  - **Example:**

```
pthread_rwlock_unlock(&lock); // Unlock the lock
```

- `pthread_rwlock_destroy()`: Destroys the lock.
  - **Syntax:**

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

  - **rwlock**: The read-write lock to destroy.
  - **Returns**: `0` on success, an error number on failure.
  - **Example:**

```
pthread_rwlock_destroy(&lock); // Destroy the read-write lock
```

Priority Inversion: It can occur when a low-priority thread holds a lock that a high-priority thread needs, leading to potential delays. Using appropriate priority settings or priority inheritance mechanisms can help mitigate this.

Example Usage:

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

```c
pthread_rwlock_t lock;
int shared_data = 0;

void* read_data(void* arg) {
    pthread_rwlock_rdlock(&lock); // Lock for reading
    printf("Thread %ld read data: %d\n", pthread_self(), shared_data);
    pthread_rwlock_unlock(&lock); // Unlock the lock
    pthread_exit(NULL); // Exit the current thread
}

void* write_data(void* arg) {
    pthread_rwlock_wrlock(&lock); // Lock for writing
    shared_data++;
    printf("Thread %ld wrote data: %d\n", pthread_self(), shared_data);
    pthread_rwlock_unlock(&lock); // Unlock the lock
    pthread_exit(NULL); // Exit the current thread
}

int main() {
    pthread_t threads[4];
    pthread_rwlock_init(&lock, NULL); // Initialize the read-write lock

    pthread_create(&threads[0], NULL, read_data, NULL); // Create read thread 1
    pthread_create(&threads[1], NULL, read_data, NULL); // Create read thread 2
    pthread_create(&threads[2], NULL, write_data, NULL); // Create write thread
    pthread_create(&threads[3], NULL, read_data, NULL); // Create read thread 3

    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL); // Wait for each thread
    }

    pthread_rwlock_destroy(&lock); // Destroy the read-write lock
    return 0;
}
```

In this example, multiple threads can read data simultaneously, but only one thread can write, ensuring data consistency.

Barriers

Purpose of Barriers

Barriers are synchronization points where threads stop until all participating threads reach the barrier. Barriers are useful in scenarios where multiple threads must synchronize at certain stages.

Basic APIs for Barrier Management:

- `pthread_barrier_init()`: Initializes a barrier.
  - **Syntax:**

    ```
    int pthread_barrier_init(pthread_barrier_t *barrier, const
    pthread_barrierattr_t *attr, unsigned count);
    ```

    - **barrier**: Pointer to the barrier to initialize.
    - **attr**: Attributes for the barrier (can be `NULL` for default attributes).
    - **count**: Number of threads that must call `pthread_barrier_wait()` before any are unblocked.
    - **Returns**: `0` on success, an error number on failure.
    - **Example:**

    ```
    pthread_barrier_t barrier;
    pthread_barrier_init(&barrier, NULL, 3); // Initialize the barrier
    for 3 threads
    ```

- `pthread_barrier_wait()`: Waits at the barrier until all threads arrive.
  - **Syntax:**

    ```
    int pthread_barrier_wait(pthread_barrier_t *barrier);
    ```

    - **barrier**: The barrier to wait at.
    - **Returns**: `PTHREAD_BARRIER_SERIAL_THREAD` for one thread or `0` for all others on success.
    - **Example:**

    ```
    pthread_barrier_wait(&barrier); // Wait at the barrier
    ```

- `pthread_barrier_destroy()`: Destroys the barrier.
  - **Syntax:**

    ```
    int pthread_barrier_destroy(pthread_barrier_t *barrier);
    ```

    - **barrier**: The barrier to destroy.
    - **Returns**: `0` on success, an error number on failure.
    - **Example:**

```
pthread_barrier_destroy(&barrier); // Destroy the barrier
```

Example Usage:

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

pthread_barrier_t barrier;

void* task(void* arg) {
    printf("Thread %ld before barrier\n", pthread_self());
    pthread_barrier_wait(&barrier); // Wait at the barrier
    printf("Thread %ld after barrier\n", pthread_self());
    pthread_exit(NULL); // Exit the current thread
}

int main() {
    pthread_t threads[3];
    pthread_barrier_init(&barrier, NULL, 3); // Initialize the barrier for 3 threads

    for (int i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, task, NULL); // Create threads
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL); // Wait for each thread
    }

    pthread_barrier_destroy(&barrier); // Destroy the barrier
    return 0;
}
```

In this example, all threads wait at the barrier until every thread reaches it, ensuring synchronized execution.

Conclusion

Threads, mutexes, semaphores, read-write locks, and barriers are crucial components for developing concurrent applications. This guide introduces how to build these synchronization primitives from scratch and provides more in-depth usage examples in C. Mastering these tools will help you write efficient, thread-safe code for complex, real-world applications.