

Mutex

1. pthread_mutex_init - [Link 1](#), [Link 2](#)

- `int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);`
- The `pthread_mutex_init()` function initializes the mutex referenced by `mutex` with attributes specified by `attr`. If `attr` is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked. [Refer](#) for `pthread_mutex_t`.

2. pthread_mutex_destroy - [Link 1](#), [Link 2](#)

- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
- The `pthread_mutex_destroy()` function shall destroy the mutex object referenced by `mutex`; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be reinitialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

3. pthread_mutex_lock - [Link 1](#), [Link 2](#)

- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- The mutex object referenced by `mutex` is locked by calling `pthread_mutex_lock()`. Mutexes are used to protect shared resources. If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it.

4. pthread_mutex_unlock - [Link 1](#), [Link 2](#)

- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- The `pthread_mutex_unlock()` function shall release the mutex object referenced by `mutex`. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by `mutex` when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

Problem 0

Write a C program to create 2 threads and update a global variable `sum` access to which is controlled by a mutex.

Solution

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

int sum = 0;
pthread_mutex_t lock;
void *solve (void *p)
{
    int val = *(int *)p;
    pthread_mutex_lock (&lock);
    //Start of Critical Section
    sum += val;
    printf ("Value: %d\n", sum);
    //End of Critical Section
    pthread_mutex_unlock (&lock);
    pthread_exit (NULL);
}

int main(int argc, char* argv[]) {
    pthread_t thread1, thread2;
    //Initialize Mutex
    if(pthread_mutex_init(&lock, NULL) != 0){
        printf("Error Initializing Mutex\n");
        return -1;
    }

    int i = 1, j = 2;
    pthread_create(&thread1, NULL, solve, &i);
    pthread_create(&thread2, NULL, solve, &j);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    //Destroy Mutex
    if(pthread_mutex_destroy(&lock) != 0){
        printf("Error Destroying Mutex\n");
        return -1;
    }
    return 0;
}
```