pthread_mutex_timedlock/pthread_mutex_timelock.c

```c
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <aio.h>
#include <math.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <sched.h>
#include <stddef.h>
#include <sys/sysinfo.h>

pthread_mutex_t mutex;
#define NUMBER_OF_TASKS 2

typedef struct
{
    int threadId;
} ThreadArgs_t;

typedef struct
{
    int period;
    int burst_time;
    int count_for_period;
    struct sched_param priority_param;
    void *(*thread_handle)(void *);
    pthread_t thread;
    ThreadArgs_t thread_args;
    void *return_Value;
    pthread_attr_t attribute;
    int target_cpu;
} RmTask_t;


typedef struct {
    double latitude;
    double longitude;
    double altitude;
    double roll;
    double pitch;
    double yaw;
    struct timespec sample_time;
} NavigationState;

NavigationState nav_state, nav_state_shouldbe;

pthread_mutex_t state_mutex;

void* update_thread(void * arg){
    struct timespec update_interval = {1, 0};
    for (int i=0; i<180; i++) {
        pthread_mutex_lock(&state_mutex);

        nav_state.latitude = i;
        nav_state.longitude = 0.5 * i;
        nav_state.altitude = 0.25 * i;
        nav_state.roll = sin(i);
        nav_state.pitch = cos(i * i);
        nav_state.yaw = cos(i);
        clock_gettime(CLOCK_REALTIME, &nav_state.sample_time);



        printf("Updated reading\n");
        printf("Yaw: %f, Roll: %f, Pitch: %f, Latitude %f, Longitude %f, Altitude %f\n",nav_state.yaw, nav_state.roll, nav_state.pitch, nav_state.latitude, nav_state.longitude, nav_state.altitude);

        // Uncomment below line to see reading thread waiting for mutex
        // sleep(10);
        pthread_mutex_unlock(&state_mutex);
        nanosleep(&update_interval, NULL);
    }
    return NULL;
}

void* read_thread(void* arg) {
    struct timespec ts;
    int s;
    struct timespec update_interval = {10, 0};
    for (int i=0; i<18; i++) { // Increased the iteration to match update_thread for continuous checking
        clock_gettime(CLOCK_REALTIME, &ts);
        ts.tv_sec += 10; // Try to acquire the lock within 10 seconds from now

        s = pthread_mutex_timedlock(&state_mutex, &ts);
        if (s == ETIMEDOUT) {
            printf("No new data available at %ld seconds\n", time(NULL));
            // No need to adjust ts because the loop will recalculate it
        } else if (s == 0) {
            // Mutex acquired, read data
            NavigationState temp_state = nav_state;
            pthread_mutex_unlock(&state_mutex);

            printf("Reading data:\n");
            printf("Yaw: %f, Roll: %f, Pitch: %f, Latitude %f, Longitude %f, Altitude %f\n",
                   temp_state.yaw, temp_state.roll, temp_state.pitch,
                   temp_state.latitude, temp_state.longitude, temp_state.altitude);
            printf("Time: tv_sec: %ld, tv_nsec: %ld\n",
                   temp_state.sample_time.tv_sec, temp_state.sample_time.tv_nsec);
        } else {
            // Handle other errors (e.g., EINVAL)
            break;
        }
        sleep(10);
    }
    return NULL;
```

```
}

void print_scheduler(void)
{
    int schedType;
    schedType = sched_getscheduler(getpid());
    switch (schedType)
    {
    case SCHED_FIFO:
        printf("Pthread Policy is SCHED_FIFO\n");
        break;
    case SCHED_OTHER:
        printf("Pthread Policy is SCHED_OTHER\n");
        break;
    case SCHED_RR:
        printf("Pthread Policy is SCHED_OTHER\n");
        break;
    default:
        printf("Pthread Policy is UNKNOWN\n");
    }
}


int main() {

    pthread_t threads[NUMBER_OF_TASKS];
    int coreid = 1;
    cpu_set_t threadcpu;

    CPU_SET(coreid, &threadcpu);

    RmTask_t tasks[NUMBER_OF_TASKS] = {
        {.period = 20,     // ms
         .burst_time = 10, // ms
         .priority_param = {1},
         .thread = threads[0],
         .thread_handle = update_thread,
         .thread_args = {0},
         .return_Value = NULL,
         .attribute = {0, 0},
         .target_cpu = 2},

        {.period = 50,
         .burst_time = 20,
         .priority_param = {2},
         .thread = threads[1],
         .thread_handle = read_thread,
         .thread_args = {0},
         .attribute = {0, 0},
         .target_cpu = 0},

    };


    pthread_attr_t attribute_flags_for_main; // for schedular type, priority
    struct sched_param main_priority_param;
```

```c

    cpu_set_t cpuset;
    int target_cpu = 1; // core we want to run our process on

    printf("This system has %d processors configured and %d processors available.\n", get_nprocs_conf(), get_nprocs());

    printf("Before adjustments to scheduling policy:\n");
    print_scheduler();

    CPU_ZERO(&cpuset); // clear all the cpus in cpuset

    int rt_max_prio = sched_get_priority_max(SCHED_FIFO);
    int rt_min_prio = sched_get_priority_min(SCHED_FIFO);

    main_priority_param.sched_priority = rt_max_prio;
    for (int i = 0; i < NUMBER_OF_TASKS; i++)
    {
        tasks[i].priority_param.sched_priority = rt_max_prio - (2*i*i);

        // initialize attributes
        pthread_attr_init(&tasks[i].attribute);

        pthread_attr_setinheritsched(&tasks[i].attribute, PTHREAD_EXPLICIT_SCHED);
        pthread_attr_setschedpolicy(&tasks[i].attribute, SCHED_FIFO);
        pthread_attr_setschedparam(&tasks[i].attribute, &tasks[i].priority_param);
        pthread_attr_setaffinity_np(&tasks[i].attribute, sizeof(cpu_set_t), &threadcpu);
    }

    pthread_attr_init(&attribute_flags_for_main);

    pthread_attr_setinheritsched(&attribute_flags_for_main, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attribute_flags_for_main, SCHED_FIFO);
    pthread_attr_setaffinity_np(&attribute_flags_for_main, sizeof(cpu_set_t), &threadcpu);

    // Main thread is already created we have to modify the priority and scheduling scheme
    int status_setting_schedular = sched_setscheduler(getpid(), SCHED_FIFO, &main_priority_param);
    if (status_setting_schedular)
    {
        printf("ERROR; sched_setscheduler rc is %d\n", status_setting_schedular);
        perror(NULL);
        exit(-1);
    }

    printf("After adjustments to scheduling policy:\n");
    print_scheduler();


    for (int i = 0; i < NUMBER_OF_TASKS; i++)
    {
        // Create a thread
        // First paramter is thread which we want to create
        // Second parameter is the flags that we want to give it to
        // third parameter is the routine we want to give
        // Fourth parameter is the value
        printf("Setting thread %d to core %d\n", i, coreid);



        if (pthread_create(&tasks[i].thread, &tasks[i].attribute, tasks[i].thread_handle, &tasks[i]) != 0)
        {
            perror("Create_Fail");
        }


    }


    for (int i = 0; i < NUMBER_OF_TASKS; i++)
    {
        pthread_join(tasks[i].thread, (void *)&tasks[i].return_Value);
    }

    if (pthread_attr_destroy(&tasks[0].attribute) != 0)
        perror("attr destroy");
    if (pthread_attr_destroy(&tasks[1].attribute) != 0)
        perror("attr destroy");
    return 0;
}
```