Tutorial(Ungraded):Pthreads
HPC

Objective : Pthreads and Synchronization

1. Dining Philosophers Problem:

The Dining Philosophers Problem is a classic synchronization problem in computer science that illustrates the challenges of resource sharing among concurrent processes. Imagine five philosophers seated around a circular table, each with a plate of food in front of them. Between every two philosophers lies a single chopstick, so there are five chopsticks in total. Each philosopher alternates between thinking and eating. While thinking, a philosopher does not require any resources and is essentially idle. However, when a philosopher decides to eat, they need to pick up both the chopstick to their left and the chopstick to their right. Because each chopstick is shared between two neighboring philosophers, a philosopher cannot eat if either chopstick is already in use by someone else.

The problem arises when multiple philosophers attempt to pick up chopsticks simultaneously. If each philosopher picks up the chopstick to their left first, all five philosophers might hold one chopstick while waiting for the second one. This creates a situation where no philosopher can proceed, resulting in a deadlock. The circular dependency among the philosophers and the limited availability of chopsticks exemplifies how competing threads can interfere with each other when accessing shared resources.

Solving the problem requires designing strategies that ensure every philosopher can eventually eat without the system becoming stuck. Common strategies include enforcing an order in which chopsticks are picked up, limiting the number of philosophers allowed to attempt to eat at the same time, or employing mechanisms that prevent circular waiting.

Solution 1 : Each philosopher (thread) picks up the left chopstick first and then the right one.

Code : click for the code

What are your observations in this code and if anything needs to be changed propose you solution to designing strategies that ensure every philosopher can eventually eat without the system becoming stuck.

2. **Problem 1: Multi-threaded Shared Counter with Mutex**
    a. **Create a Simple Threaded Counter (With Race Condition)**
    b. **Fix the Race Condition Using Mutex**
3. **Parallel Merge Sort using Pthreads.**

Others:

4. **Convert the following code to parallelize using pThreads**

```
int arr[N], prefix_sum[N];

void compute_prefix_sum() {

  prefix_sum[0] = arr[0];

  for (int i = 1; i < N; i++) {

    prefix_sum[i] = prefix_sum[i - 1] + arr[i];

  }

}
```

5. **What is expected output**

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

int counter = 0;

void *increment(void *arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Final counter value: %d\n", counter);
    return 0;
}
```

6.

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

pthread_mutex_t lock1, lock2;

void *thread1_func(void *arg) {
    pthread_mutex_lock(&lock1);
    pthread_mutex_lock(&lock2);
    printf("Thread 1 executing\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}

void *thread2_func(void *arg) {
    pthread_mutex_lock(&lock2);
    pthread_mutex_lock(&lock1);
    printf("Thread 2 executing\n");
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);
    pthread_create(&t1, NULL, thread1_func, NULL);
    pthread_create(&t2, NULL, thread2_func, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_mutex_destroy(&lock1);
    pthread_mutex_destroy(&lock2);
    return 0;
}
```

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

#define MAX_THREADS 16
#define N 100000000
long long *arr;
long long sum[MAX_THREADS] = {0};
int num_threads = 1;
typedef struct {
    int start;
    int end;
    int thread_id;
} ThreadData;

void* sum_array(void* arg) {
    ThreadData *data = (ThreadData*)arg;
    long long local_sum = 0;

    for (int i = data->start; i < data->end; i++) {
        local_sum += arr[i];
    }

    sum[data->thread_id] = local_sum;
    pthread_exit(NULL);
}

double Time_T(int threads) {
    pthread_t thread[threads];
    ThreadData thread_data[threads];

    clock_t start_time = clock();

    int chunk_size = N / threads;
    for (int i = 0; i < threads; i++) {
        thread_data[i].start = i * chunk_size;
        thread_data[i].end = (i == threads - 1) ? N : (i +
1) * chunk_size;
        thread_data[i].thread_id = i;
        pthread_create(&thread[i], NULL, sum_array,
(void*)&thread_data[i]);
    }
```

```
    // Join threads and compute final sum
    long long final_sum = 0;
    for (int i = 0; i < threads; i++) {
        pthread_join(thread[i], NULL);
        final_sum += sum[i];
    }

    clock_t end_time = clock();
    return ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
}

int main() {
    // Allocate memory for array
    arr = (long long*)malloc(N * sizeof(long long));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

        for (int i = 0; i < N; i++) {
        arr[i] = 1;

    }

    for (int t = 1; t <= MAX_THREADS; t++) {
        double time = Time_T(t);
        printf("%d, %lf\n", t, time);
    }

    free(arr);


            }

    free(arr);
    return 0;
}
```