Exercise: Implementing Multi-threaded MergeSort in C

Objective:

Learn how to implement and use threads in C by parallelizing the MergeSort algorithm.

Description:

MergeSort is a divide-and-conquer algorithm that divides the array into halves, sorts each half, and then merges the sorted halves. In this exercise, you will implement a multi-threaded version of MergeSort in C using POSIX threads (pthreads). Each thread will handle sorting a segment of the array. After sorting, you will merge the segments in a correct hierarchical manner to preserve sorting correctness.

Requirements:

- Implement the MergeSort algorithm.
- Use pthreads to create multiple threads for sorting different parts of the array.
- Merge the sorted segments in a single-threaded manner.
- Measure and compare the performance of the multi-threaded version with the single-threaded version.

Steps:

1. Single-threaded MergeSort Implementation: Implement the basic single-threaded MergeSort algorithm.
2. Multi-threaded MergeSort Implementation: Create a structure to pass parameters to the thread function. Implement the thread function that performs MergeSort on a segment of the array. Create threads and assign each thread a segment of the array to sort. Merge the sorted segments after all threads have completed.
3. Merging Strategy: merge the segments **hierarchically** in pairs, as done in the original recursive MergeSort:
   A + B → AB
   C + D → CD
   AB + CD → ABCD

4. Performance Measurement: Measure the time taken by the single-threaded and multi-threaded versions.

Additional Explanation:

By using multi-threading, the MergeSort algorithm can sort large arrays more efficiently. When the array size (n) is large, you will observe a significant improvement in performance with the multi-threaded version compared to the single-threaded version. This is because the sorting work is divided among multiple threads, allowing parallel processing and better utilization of multi-core processors.

Code Template:

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

#define MAX_THREADS 4

typedef struct {
   int *array;
   int left;
   int right;
} ThreadData;

void merge(int arr[], int left, int mid, int right) {
   // Merging logic
}

void mergeSort(int arr[], int left, int right) {
   if (left < right) {
      int mid = left + (right - left) / 2;
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);
      merge(arr, left, mid, right);
   }
}

void *threadedMergeSort(void *arg) {
   ThreadData *data = (ThreadData *)arg;
   mergeSort(data->array, data->left, data->right);
   pthread_exit(NULL);
}
```

```
// Hierarchical merging of sorted segments

void hierarchicalMerge(int arr[], ThreadData segments[], int count) {

        // TODO: Implement hierarchical merge of segments

}

void multiThreadedMergeSort(int arr[], int n) {
  pthread_t threads[MAX_THREADS];
  ThreadData threadData[MAX_THREADS];
  int segmentSize = n / MAX_THREADS;

  // Create threads for each segment
  for (int i = 0; i < MAX_THREADS; i++) {
    threadData[i].array = arr;
    threadData[i].left = i * segmentSize;
    threadData[i].right = (i == MAX_THREADS - 1) ? n - 1 : (i + 1) * segmentSize - 1;
    pthread_create(&threads[i], NULL, threadedMergeSort, &threadData[i]);
  }

  // Join threads
  for (int i = 0; i < MAX_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }


// Merge the sorted segments hierarchically

  hierarchicalMerge(arr, threadData, MAX_THREADS);
}

void printArray(int arr[], int size) {
  for (int i = 0; i < size; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
}

int main() {
  int n = 16; // Change size as needed
  int arr[n];
  srand(time(0));

  for (int i = 0; i < n; i++) {
```

```
        arr[i] = rand() % 100;
    }

    printf("Original array: \n");
    printArray(arr, n);

    clock_t start, end;
    double cpu_time_used;

    // Single-threaded MergeSort
    start = clock();
    mergeSort(arr, 0, n-1);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Sorted array (Single-threaded): \n");
    printArray(arr, n);
    printf("Time taken by single-threaded MergeSort: %f seconds\n", cpu_time_used);

    // Generate a new random array
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    printf("Original array: \n");
    printArray(arr, n);

    // Multi-threaded MergeSort
    start = clock();
    multiThreadedMergeSort(arr, n);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Sorted array (Multi-threaded): \n");
    printArray(arr, n);
    printf("Time taken by multi-threaded MergeSort: %f seconds\n", cpu_time_used);

    return 0;
}
```

This exercise helps in understanding thread creation, synchronization, and the performance benefits of multi-threading.

Enjoy!