1. Develop a program using pthread to concatenate multiple strings passed to thread function.

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

#define MAX_LEN 1024

void* concatenate_strings(void *arg) {

  char **strings = (char **)arg;

  char *result = (char *)malloc(MAX_LEN * sizeof(char));

  result[0] = '\0';

  for (int i = 0; strings[i] != NULL; i++) {

    strcat(result, strings[i]);

  }

  return result;

}

int main() {

  pthread_t thread;

  // Strings to concatenate

  char *strings[] = {"Hello, ", "world", "! ", "Welcome to ", "multithreading.", NULL};

  // Create the thread to concatenate the strings

  pthread_create(&thread, NULL, concatenate_strings, (void *)strings);

  // Wait for the thread to finish and get the result

  char *result;

  pthread_join(thread, (void **)&result);

  // Print the concatenated string

  printf("Concatenated String: %s\n", result);
```

```
    // Free the allocated memory for result

    free(result);

    return 0;

}
```

2. Create a pthread program to find length of strings passed to the thread function.

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

#define NUM_STRINGS 4

void* find_length(void *arg) {

    char *string = (char *)arg;

    int *length = (int *)malloc(sizeof(int));

    *length = strlen(string);

    return length;

}

int main() {

    pthread_t threads[NUM_STRINGS];

    char *strings[NUM_STRINGS] = {"Hello", "world", "pthread", "example"};

    int *lengths[NUM_STRINGS];

    for (int i = 0; i < NUM_STRINGS; i++) {

        pthread_create(&threads[i], NULL, find_length, (void *)strings[i]);

    }

    for (int i = 0; i < NUM_STRINGS; i++) {

        pthread_join(threads[i], (void **)&lengths[i]);

        printf("Length of string \"%s\": %d\n", strings[i], *lengths[i]);
        free(lengths[i]);
    }
    return 0;
}
```

3. Implement a program that performs statistical operations (calculating average, maximum and minimum) for a set of numbers. Utilize three threads, where each thread performs its respective operation.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_COUNT 6
typedef struct {
    int *numbers;
    int count;
    double result;
} thread_data;
// Thread function to calculate the average
void* calculate_average(void *arg) {
    thread_data *data = (thread_data *)arg;
    int sum = 0;
    for (int i = 0; i < data->count; i++) {
        sum += data->numbers[i];
    }
    data->result = (double)sum / data->count;
    return NULL;
}
// Thread function to find the maximum
```

```
void* find_maximum(void *arg) {
    thread_data *data = (thread_data *)arg;
    int max = data->numbers[0];
    for (int i = 1; i < data->count; i++) {
        if (data->numbers[i] > max) {
            max = data->numbers[i];
        }
    }
    data->result = max;
    return NULL;
}
// Thread function to find the minimum
void* find_minimum(void *arg) {
    thread_data *data = (thread_data *)arg;
    int min = data->numbers[0];
    for (int i = 1; i < data->count; i++) {
        if (data->numbers[i] < min) {
            min = data->numbers[i];
        }
    }
    data->result = min;
    return NULL;
}
int main() {
    pthread_t threads[3];
    int numbers[NUM_COUNT] = {10, 20, 5, 40, 25, 15};
    thread_data avg_data = {numbers, NUM_COUNT, 0};
    thread_data max_data = {numbers, NUM_COUNT, 0};
    thread_data min_data = {numbers, NUM_COUNT, 0};
    pthread_create(&threads[0], NULL, calculate_average, &avg_data);
    pthread_create(&threads[1], NULL, find_maximum, &max_data);
    pthread_create(&threads[2], NULL, find_minimum, &min_data);
    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Average: %.2f\n", avg_data.result);
    printf("Maximum: %.0f\n", max_data.result);
    printf("Minimum: %.0f\n", min_data.result);
    return 0;
}
```

4. Write a multithreaded program where a globally passed array of integers is divided into two smaller lists and given as input to two threads. Each thread sorts their half of list and then passes the sorted lists to third thread, which merges and sorts them. The final sorted list is printed by the parent thread.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define ARRAY_SIZE 10
int array[ARRAY_SIZE] = {38, 27, 43, 3, 9, 82, 10, 15, 6, 20};
int sorted_array[ARRAY_SIZE];
typedef struct {
    int *array;
    int size;
} thread_data;
void merge(int *left, int left_size, int *right, int right_size, int *merged) {
    int i = 0, j = 0, k = 0;
    while (i < left_size && j < right_size) {
        if (left[i] < right[j]) {
            merged[k++] = left[i++];
        } else {
            merged[k++] = right[j++];
        }
    }
    while (i < left_size) {
        merged[k++] = left[i++];
    }
    while (j < right_size) {
        merged[k++] = right[j++];
    }
}
int compare(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}
void* sort_half(void *arg) {
    thread_data *data = (thread_data *)arg;
    qsort(data->array, data->size, sizeof(int), compare);
    return NULL;
}
void* merge_sorted_arrays(void *arg) {
    int mid = ARRAY_SIZE / 2;
    merge(array, mid, array + mid, ARRAY_SIZE - mid, sorted_array);
    return NULL;
```

```
}
int main() {
    pthread_t thread1, thread2, thread3;
    thread_data data1 = {array, ARRAY_SIZE / 2};
    thread_data data2 = {array + ARRAY_SIZE / 2, ARRAY_SIZE - ARRAY_SIZE / 2};
    pthread_create(&thread1, NULL, sort_half, &data1);
    pthread_create(&thread2, NULL, sort_half, &data2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_create(&thread3, NULL, merge_sorted_arrays, NULL);
    pthread_join(thread3, NULL);
    printf("Sorted array: ");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", sorted_array[i]);
    }
    printf("\n");
    return 0;}
```

5. Create a program using pthread_create to generate multiple threads. Each thread should display its unique ID and execution sequence.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 5
void* thread_function(void *arg) {
    int thread_num = *(int *)arg;
    printf("Thread %d: Unique ID = %lu\n", thread_num, pthread_self());
    return NULL;
}
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_args[i] = i + 1;
        pthread_create(&threads[i], NULL, thread_function, (void *)&thread_args[i]);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

6. Create a threaded application that demonstrates graceful thread termination using pthread_exit for resource cleanup compared to abrupt termination via pthread_cancel.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
void* graceful_termination(void *arg) {
    printf("Graceful Thread: Starting execution.\n");
    for (int i = 0; i < 5; i++) {
        printf("Graceful Thread: Working... %d\n", i + 1);
        sleep(1); // Simulate some work
    }
    printf("Graceful Thread: Cleaning up resources.\n");
    pthread_exit(NULL); // Graceful exit
}
void* abrupt_termination(void *arg) {
    printf("Abrupt Thread: Starting execution.\n");
    for (int i = 0; i < 10; i++) {
        printf("Abrupt Thread: Working... %d\n", i + 1);
        sleep(1); // Simulate some work
    }
    printf("Abrupt Thread: Should never get here if canceled.\n");
    pthread_exit(NULL);
}
int main() {
    pthread_t thread1, thread2;
    // Create first thread with graceful termination
    pthread_create(&thread1, NULL, graceful_termination, NULL);
    // Create second thread with abrupt termination
    pthread_create(&thread2, NULL, abrupt_termination, NULL);
    // Wait for a few seconds to let the threads start their work
    sleep(3);
    // Cancel the second thread abruptly
    printf("Main Thread: Canceling Abrupt Thread.\n");
    pthread_cancel(thread2);
    // Wait for the threads to complete
    pthread_join(thread1, NULL);  // Graceful thread completes normally
    pthread_join(thread2, NULL);  // Abrupt thread may not complete work
    printf("Main Thread: All threads joined.\n");
    return 0;}
```