TASK 1:

This program decrypts cryptographic password hashes using multiple threads. The main function distributes the workload among the threads. Each thread is given a range of password combinations to check. To decrypt the password, the program goes through possible combinations of letters and numbers. It tries to match the encrypted password with the combinations. Mutex locks are used to deal with any potential issues when updating the global variable that holds the discovered password. If a match is found, the program prints the decrypted password along with relevant details. This program provides an efficient way to decrypt password hashes in parallel.

1. Cracks a password using multithreading and dynamic slicing based on thread count:

The program dynamically distributes the workload of password cracking among multiple threads based on the specified thread count. It calculates the workload for each thread to ensure an even distribution of combinations to check. Threads operate concurrently to search for the password within their assigned ranges.

```
// Number of threads to utilize
int threadCount = 6;

// Array for storing thread IDs
pthread_t *threadIDs = malloc(sizeof(pthread_t) * threadCount);

// Total number of password combinations to check
int totalPasswordCombinations = 26 * 26 * 100;

// Calculate workload distribution per thread
int remainder = totalPasswordCombinations % threadCount;
totalPasswordCombinations = totalPasswordCombinations - remainder;
int workPerThread = totalPasswordCombinations / threadCount;
int threadCorrection = 0;

// Create threads
for (int i = 0; i < threadCount; i++)
{
    // Allocate memory for thread-specific arguments
```

```
        struct ThreadParameters *currentThreadParameters = malloc(sizeof(struct
ThreadParameters));

        // Initialize thread-specific arguments
        currentThreadParameters->threadIndex = i;
        currentThreadParameters->encryptedPassword = targetEncryptedPassword;
        currentThreadParameters->start = threadCorrection;
        threadCorrection = threadCorrection + workPerThread;

        // Adjust workload distribution for any remainder
        if (remainder > 0)
        {
            currentThreadParameters->end = threadCorrection;
            threadCorrection += 1;
            remainder -= 1;
        }
        else
        {
            currentThreadParameters->end = threadCorrection - 1;
        }

        // Display the range of combinations each thread will handle
        printf("Thread '%d' will handle the range from '%d' to '%d'\n", i,
currentThreadParameters->start, currentThreadParameters->end);

        // Create a thread and pass thread-specific arguments
        pthread_create(&threadIDs[i], NULL, Decrypt, currentThreadParameters);
    }

    // Wait for all threads to finish
    for (int i = 0; i < threadCount; i++)
    {
        pthread_join(threadIDs[i], NULL);
    }
    // Display results if a password is found
    if (DiscoveredPassword != NULL)
    {
        printf("---\nResults:\n");
        printf("Encrypted Password: '%s'\n", targetEncryptedPassword);
        printf("Decrypted Password: '%s'\n", DiscoveredPassword);
        free(DiscoveredPassword);
    }
    // Free allocated memory
    free(threadIDs);
}
```

2. Program finishes appropriately when password has been found:

The program utilizes a global variable, DiscoveredPassword, to store the password once it has been found. Threads check this variable before continuing their search, and if the password has been discovered by another thread, they exit gracefully. The main program displays the results appropriately based on whether the password has been found or not.

```
// Function for decrypting passwords within a specified range
void *Decrypt(void *threadParameters)
{
    struct ThreadParameters *params = (struct ThreadParameters
*)threadParameters;
    char salt[7];
    substring(salt, params->encryptedPassword, 0, 6);
    char potentialPassword[7];
    char *encrypted;
    struct crypt_data data;
    data.initialized = 0;

    // Iterate through the assigned range of password combinations
    for (int i = params->start; i <= params->end; i++)
    {
        // Exit the thread if a password has already been found by another thread
        if (DiscoveredPassword != NULL)
        {
            pthread_exit(NULL);
        }

        // Calculate indices for the current combination
        int secondThirdPossibilities = 26 * 100;
        int firstAlphabetIndex = i / secondThirdPossibilities;
        char firstChar = (char)(firstAlphabetIndex + 'A');
        int secondAlphabetIndex = (i / 100) % 26;
        char secondChar = (char)(secondAlphabetIndex + 'A');
        int numbers = i % 100;

        // Display the current combination being checked by the thread
        printf("Thread '%d' Combination: '%c %c %02d'\n", params->threadIndex,
firstChar, secondChar, numbers);

        // Formulate the potential password
        sprintf(potentialPassword, "%c%c%02d", firstChar, secondChar, numbers);
        encrypted = (char *)crypt_r(potentialPassword, salt, &data);
```

```
        // Check if the potential password matches the target encrypted password
        if (strcmp(params->encryptedPassword, encrypted) == 0)
        {
            pthread_mutex_lock(&mutex);
            printf("Thread '%d' has determined the password to be '%s'!\n",
params->threadIndex, potentialPassword);
            DiscoveredPassword = strdup(potentialPassword);
            pthread_mutex_unlock(&mutex);
            pthread_exit(NULL);
        }
    }

    // Free thread-specific arguments before exiting the thread
    free(params);
    pthread_exit(NULL);
}
```