Sure, I can help you modify your code to perform parallel encryption using multiple threads. To achieve this, we'll need to use a threading library like POSIX threads (pthread). I'll outline the steps you need to take to implement parallel encryption using threads:

1. \*\*Include Necessary Headers\*\*:

Include the necessary headers for using pthreads and other required libraries.

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <unistd.h>

#include <dirent.h>

#include <openssl/sha.h>

#include <openssl/ssl.h>

// Include other necessary headers

```

2. \*\*Create a Thread Function for Encryption\*\*:

Define a function that performs the encryption of a single file. This function will be executed in multiple threads.

```c

void\* encrypt\_thread(void\* arg) {

// Extract necessary data from the argument

// Call aes\_encrypt or other encryption functions

// Handle other operations related to encryption

return NULL;

}

```

3. \*\*Modify Main Logic for Parallel Encryption\*\*:

In your main function or wherever the file handling logic is, create and manage threads for parallel encryption.

```c

int main() {

// Initialize necessary variables and data structures

pthread\_t threads[6]; // You can change the number of threads as needed

// Loop to create and start threads

for (int i = 0; i < 6; i++) {

// Create a data structure to hold thread-specific data

// Initialize the data structure with necessary information

// Example: struct ThreadData data = { ... };

// Create the thread

if (pthread\_create(&threads[i], NULL, encrypt\_thread, &data) != 0) {

perror("pthread\_create");

// Handle error

}

}

// Wait for threads to finish

for (int i = 0; i < 6; i++) {

if (pthread\_join(threads[i], NULL) != 0) {

perror("pthread\_join");

// Handle error

}

}

// Clean up and exit

return 0;

}

```

4. \*\*Update Thread Function\*\*:

Modify the `encrypt\_thread` function to perform the encryption for a single file.

```c

void\* encrypt\_thread(void\* arg) {

// Extract necessary data from the argument

struct ThreadData\* data = (struct ThreadData\*)arg;

// Call aes\_encrypt or other encryption functions using data

ret = aes\_encrypt(data->input\_file, data->output\_file, data->str, data->hash);

// Handle other operations related to encryption

return NULL;

}

```

Remember to properly define and populate the `ThreadData` structure with the necessary information for each thread, such as input and output file paths, `str`, and `hash`.

Additionally, you will need to ensure proper synchronization mechanisms, such as mutexes or semaphores, if the threads access shared resources concurrently.

Please note that threading involves complex synchronization and resource management, and debugging threaded code can be challenging. Make sure to test thoroughly and handle errors gracefully.

Certainly! Here's a modified version of your encryption code with the necessary modifications to implement parallel encryption using pthreads. Note that this example includes the required modifications, but you'll need to adapt it to your specific code structure and include the missing parts (like function definitions and shared resource management) from your original code.

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <openssl/sha.h>

#include <unistd.h>

// Include other necessary headers

// Define the ThreadData structure to hold thread-specific data

struct ThreadData {

// Add necessary data fields here, like input\_file, output\_file, str, hash, etc.

};

// Function to perform encryption in a thread

void\* encrypt\_thread(void\* arg) {

struct ThreadData\* data = (struct ThreadData\*)arg;

// Call aes\_encrypt or other encryption functions using data

int ret = aes\_encrypt(data->input\_file, data->output\_file, data->str, data->hash);

// Handle other operations related to encryption

return NULL;

}

int main() {

// Initialize necessary variables and data structures

pthread\_t threads[6]; // You can change the number of threads as needed

struct ThreadData thread\_data[6]; // Data for each thread

// Loop to create and start threads

for (int i = 0; i < 6; i++) {

// Initialize thread\_data[i] with necessary information

// thread\_data[i].input\_file = ...;

// thread\_data[i].output\_file = ...;

// thread\_data[i].str = ...;

// thread\_data[i].hash = ...;

// Create the thread

if (pthread\_create(&threads[i], NULL, encrypt\_thread, &thread\_data[i]) != 0) {

perror("pthread\_create");

// Handle error

}

}

// Wait for threads to finish

for (int i = 0; i < 6; i++) {

if (pthread\_join(threads[i], NULL) != 0) {

perror("pthread\_join");

// Handle error

}

}

// Clean up and exit

return 0;

}

```

Remember to adapt this code to your specific use case and integrate it into your existing codebase. You will need to provide the necessary definitions and implementations for functions like `aes\_encrypt`, proper data initialization in `thread\_data`, and any required synchronization mechanisms.

Please test the code thoroughly to ensure correctness and proper concurrency handling. Threading can be complex, so make sure to address any synchronization issues that may arise.

Certainly! You would need to replace the comments with the actual values that you want to provide to each thread for encryption. Here's how you might initialize the `thread\_data` structure with the necessary information for each thread:

```c

// Loop to create and start threads

for (int i = 0; i < 6; i++) {

// Initialize thread\_data[i] with necessary information

thread\_data[i].input\_file = "input\_file\_path"; // Provide the actual input file path

thread\_data[i].output\_file = "output\_file\_path"; // Provide the actual output file path

thread\_data[i].str = "data\_str"; // Provide the actual data string

thread\_data[i].hash = "hash\_value"; // Provide the actual hash value

// Create the thread

if (pthread\_create(&threads[i], NULL, encrypt\_thread, &thread\_data[i]) != 0) {

perror("pthread\_create");

// Handle error

}

}

```

Replace `"input\_file\_path"`, `"output\_file\_path"`, `"data\_str"`, and `"hash\_value"` with the actual values you want to provide to each thread for encryption.

Make sure to update the `ThreadData` structure definition to include the appropriate data types for `input\_file`, `output\_file`, `str`, and `hash`. This will ensure that the provided values are properly stored in the structure and accessible within the `encrypt\_thread` function.

Apologies for any misunderstanding, but the code I provided in the previous responses doesn't inherently perform parallel processing of six files simultaneously. It sets up six threads, and each thread performs encryption on a single file, one after the other, due to the nature of how the loop is constructed.

If you want to achieve parallel processing of multiple files simultaneously, you will need to design your code in a way that allows threads to work on multiple files concurrently. To achieve this, you can implement a thread pool or a queue-based approach where threads pick up work (files) as they become available.

Here's a high-level outline of how you could approach it:

1. \*\*Initialize Thread Pool\*\*: Create a pool of threads that are ready to process files concurrently.

2. \*\*Create a Work Queue\*\*: Maintain a queue of files that need to be encrypted. Whenever a new file needs to be encrypted, add it to this queue.

3. \*\*Thread Function\*\*: Each thread in the pool should loop indefinitely, trying to dequeue files from the work queue. Once a file is dequeued, the thread performs the encryption.

4. \*\*Main Logic\*\*: Your main logic should enqueue files into the work queue and manage the threads in the thread pool.

Here's a simplified example of what this approach could look like:

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

// Include other necessary headers

#define NUM\_THREADS 6

#define FILE\_QUEUE\_SIZE 10

typedef struct {

// Define fields for your thread-specific data here

} ThreadData;

pthread\_mutex\_t queue\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t queue\_cond = PTHREAD\_COND\_INITIALIZER;

char\* file\_queue[FILE\_QUEUE\_SIZE];

int queue\_front = 0;

int queue\_rear = 0;

void\* encrypt\_thread(void\* arg) {

while (1) {

pthread\_mutex\_lock(&queue\_mutex);

while (queue\_front == queue\_rear) {

pthread\_cond\_wait(&queue\_cond, &queue\_mutex);

}

char\* file\_path = file\_queue[queue\_front];

queue\_front = (queue\_front + 1) % FILE\_QUEUE\_SIZE;

pthread\_mutex\_unlock(&queue\_mutex);

// Perform encryption on file\_path

// ...

free(file\_path);

}

return NULL;

}

int main() {

// Initialize threads

pthread\_t threads[NUM\_THREADS];

for (int i = 0; i < NUM\_THREADS; i++) {

pthread\_create(&threads[i], NULL, encrypt\_thread, NULL);

}

// Add files to the work queue

for (int i = 0; i < number\_of\_files; i++) {

char\* file\_path = strdup("file\_path"); // Replace with actual file path

pthread\_mutex\_lock(&queue\_mutex);

if ((queue\_rear + 1) % FILE\_QUEUE\_SIZE != queue\_front) {

file\_queue[queue\_rear] = file\_path;

queue\_rear = (queue\_rear + 1) % FILE\_QUEUE\_SIZE;

pthread\_cond\_signal(&queue\_cond);

} else {

// Handle queue full

free(file\_path);

}

pthread\_mutex\_unlock(&queue\_mutex);

}

// Join threads

for (int i = 0; i < NUM\_THREADS; i++) {

pthread\_join(threads[i], NULL);

}

return 0;

}

```

Please note that this is a simplified example, and you might need to adapt it to fit your specific code structure and requirements. This approach allows for concurrent processing of files using a thread pool and a work queue. Each thread picks up a file from the queue and processes it, allowing for multiple files to be encrypted simultaneously.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <openssl/sha.h>

#include <unistd.h>

// Include other necessary headers

// Define the ThreadData structure to hold thread-specific data

struct ThreadData {

const char\* input\_file;

const char\* output\_file;

const char\* str;

const char\* hash;

};

// Function to perform encryption in a thread

void\* encrypt\_thread(void\* arg) {

struct ThreadData\* data = (struct ThreadData\*)arg;

// Call aes\_encrypt or other encryption functions using data

int ret = aes\_encrypt(data->input\_file, data->output\_file, data->str, data->hash);

// Handle other operations related to encryption

return NULL;

}

int main() {

// Initialize necessary variables and data structures

pthread\_t threads[6]; // You can change the number of threads as needed

struct ThreadData thread\_data[6]; // Data for each thread

// Set thread\_data for each thread

// Initialize thread\_data[i] with necessary information

thread\_data[0].input\_file = "file1\_input"; // Provide the actual input file path

thread\_data[0].output\_file = "file1\_output"; // Provide the actual output file path

thread\_data[0].str = "data\_str\_1"; // Provide the actual data string

thread\_data[0].hash = "hash\_value\_1"; // Provide the actual hash value

// Set data for other threads similarly...

// Loop to create and start threads

for (int i = 0; i < 6; i++) {

// Create the thread

if (pthread\_create(&threads[i], NULL, encrypt\_thread, &thread\_data[i]) != 0) {

perror("pthread\_create");

// Handle error

}

}

// Wait for threads to finish

for (int i = 0; i < 6; i++) {

if (pthread\_join(threads[i], NULL) != 0) {

perror("pthread\_join");

// Handle error

}

}

// Clean up and exit

return 0;

}

To perform parallel encryption of multiple files using multiple threads, you can utilize the threading capabilities provided by the C programming language. One common threading library is the POSIX threads library (`pthread`), which allows you to create and manage threads. I'll provide you with an outline of how you could modify your code to achieve parallel encryption using multiple threads:

1. \*\*Include Necessary Headers\*\*: Include the necessary headers for threading and other functions you'll use.

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <unistd.h>

#include "your\_encryption\_functions.h" // Include your encryption functions

```

2. \*\*Define Thread Function\*\*: Create a function that will serve as the entry point for each thread. This function will perform the encryption for a specific file.

```c

void \*thread\_function(void \*arg) {

// Cast the argument back to the appropriate data type

ThreadArgs \*args = (ThreadArgs \*)arg;

// Perform encryption for the specified file

encrypt\_file(args->source\_file, args->hash);

// Clean up and exit the thread

free(args);

pthread\_exit(NULL);

}

```

3. \*\*Main Function with Thread Creation\*\*: Modify your main function to create and manage threads for parallel encryption.

```c

int main() {

// ... Your other code ...

// Create an array of thread IDs and thread arguments

pthread\_t threads[NUM\_THREADS];

ThreadArgs \*thread\_args[NUM\_THREADS];

// Loop to create threads

for (int i = 0; i < NUM\_THREADS; i++) {

// Create thread arguments

ThreadArgs \*args = (ThreadArgs \*)malloc(sizeof(ThreadArgs));

if (args == NULL) {

fprintf(stderr, "Memory allocation error\n");

exit(EXIT\_FAILURE);

}

args->source\_file = ...; // Set the source file for this thread

args->hash = ...; // Set the hash for this thread

// Create the thread

int ret = pthread\_create(&threads[i], NULL, thread\_function, (void \*)args);

if (ret != 0) {

fprintf(stderr, "Thread creation error\n");

exit(EXIT\_FAILURE);

}

}

// Wait for all threads to finish

for (int i = 0; i < NUM\_THREADS; i++) {

pthread\_join(threads[i], NULL);

}

// ... Your other code ...

}

```

Please note that this is just an outline and you will need to integrate this threading logic into your existing code properly. Replace `your\_encryption\_functions.h` with the header file that contains the implementation of your `encrypt\_file` function.

Remember to handle memory allocation and deallocation properly to avoid memory leaks. Also, ensure that shared resources (like the `output\_file` and the data structures accessed by threads) are properly synchronized to avoid race conditions. You might need to use synchronization mechanisms like mutexes or semaphores if your threads are accessing shared data concurrently.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_THREADS 6

typedef struct {

char \*source\_file;

char \*hash;

} ThreadArgs;

void \*thread\_function(void \*arg) {

ThreadArgs \*args = (ThreadArgs \*)arg;

// Simulate encryption

printf("Thread: Encrypting %s with hash %s\n", args->source\_file, args->hash);

// Clean up and exit the thread

free(args);

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS];

ThreadArgs \*thread\_args[NUM\_THREADS];

// Sample data for demonstration

const char \*source\_files[NUM\_THREADS] = {

"file1.txt", "file2.txt", "file3.txt", "file4.txt", "file5.txt", "file6.txt"

};

const char \*hashes[NUM\_THREADS] = {

"hash1", "hash2", "hash3", "hash4", "hash5", "hash6"

};

for (int i = 0; i < NUM\_THREADS; i++) {

ThreadArgs \*args = (ThreadArgs \*)malloc(sizeof(ThreadArgs));

if (args == NULL) {

fprintf(stderr, "Memory allocation error\n");

exit(EXIT\_FAILURE);

}

args->source\_file = source\_files[i];

args->hash = hashes[i];

thread\_args[i] = args;

int ret = pthread\_create(&threads[i], NULL, thread\_function, (void \*)args);

if (ret != 0) {

fprintf(stderr, "Thread creation error\n");

exit(EXIT\_FAILURE);

}

}

for (int i = 0; i < NUM\_THREADS; i++) {

pthread\_join(threads[i], NULL);

}

return 0;

}

Certainly! I'll provide you with a modified version of your code that includes the use of mutexes for synchronization between threads. Since you mentioned that you want to perform parallel encryption of multiple files using six threads, I'll integrate mutexes to ensure proper synchronization while accessing shared resources.

Please note that this code is an adaptation based on the provided snippets, and you may need to adjust it to match the actual structure and implementation of your program.

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include "your\_encryption\_functions.h" // Include your encryption functions

#define NUM\_THREADS 6

pthread\_mutex\_t mutex; // Mutex for synchronization

int files\_remaining = NUM\_THREADS; // Counter to track files remaining to process

struct ThreadArgs {

char \*source\_file;

char \*hash;

};

void \*thread\_function(void \*arg) {

ThreadArgs \*args = (ThreadArgs \*)arg;

// Perform encryption for the specified file

encrypt\_file(args->source\_file, args->hash);

// Lock mutex to update files\_remaining counter

pthread\_mutex\_lock(&mutex);

files\_remaining--;

pthread\_mutex\_unlock(&mutex);

free(args); // Clean up

pthread\_exit(NULL);

}

int main() {

// Initialize mutex

if (pthread\_mutex\_init(&mutex, NULL) != 0) {

fprintf(stderr, "Mutex initialization failed\n");

exit(EXIT\_FAILURE);

}

// ... Your other code ...

// Create threads

pthread\_t threads[NUM\_THREADS];

ThreadArgs \*thread\_args[NUM\_THREADS];

for (int i = 0; i < NUM\_THREADS; i++) {

ThreadArgs \*args = (ThreadArgs \*)malloc(sizeof(ThreadArgs));

if (args == NULL) {

fprintf(stderr, "Memory allocation error\n");

exit(EXIT\_FAILURE);

}

args->source\_file = ...; // Set the source file for this thread

args->hash = ...; // Set the hash for this thread

thread\_args[i] = args; // Store thread arguments

if (pthread\_create(&threads[i], NULL, thread\_function, (void \*)args) != 0) {

fprintf(stderr, "Thread creation error\n");

exit(EXIT\_FAILURE);

}

}

// Wait for all threads to finish

for (int i = 0; i < NUM\_THREADS; i++) {

pthread\_join(threads[i], NULL);

}

// Clean up mutex

pthread\_mutex\_destroy(&mutex);

// ... Your other code ...

}

```

In this modified code:

1. A mutex `mutex` is initialized to control access to the shared resources (like the `files\_remaining` counter).

2. Each thread, before performing encryption, locks the mutex to update the `files\_remaining` counter. This ensures that the counter is properly updated and accessed by only one thread at a time.

3. The `files\_remaining` counter is decremented after a file is processed by a thread.

4. Threads clean up allocated memory and exit using `pthread\_exit`.

Remember that you need to fill in the appropriate source file and hash values for each thread. Also, ensure that you have your encryption functions properly implemented (`encrypt\_file`, etc.). This example is designed to show the usage of mutexes for synchronization; you might need to adapt it to your complete code structure.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#define NUM\_THREADS 6

// Define your data structures, constants, and functions here

// Mutex for synchronization

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

// Thread function for encryption

void \*encrypt\_thread(void \*arg) {

// Cast the argument to a structure containing necessary data

// (e.g., input file, output file, data.str, and hash)

// TODO: Perform encryption

// Lock the mutex before writing to the output file

pthread\_mutex\_lock(&mutex);

// TODO: Write encrypted data to the output file

// Unlock the mutex

pthread\_mutex\_unlock(&mutex);

// TODO: Return appropriate value

return NULL;

}

int main() {

// Initialize your variables and data structures

// Create an array of thread IDs

pthread\_t threads[NUM\_THREADS];

// Create mutex

if (pthread\_mutex\_init(&mutex, NULL) != 0) {

fprintf(stderr, "Mutex initialization failed\n");

return 1;

}

// Create threads for parallel encryption

for (int i = 0; i < NUM\_THREADS; ++i) {

// TODO: Set up thread data

if (pthread\_create(&threads[i], NULL, encrypt\_thread, (void \*)&thread\_data[i]) != 0) {

fprintf(stderr, "Thread creation failed\n");

return 1;

}

}

// Wait for all threads to finish

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_join(threads[i], NULL);

}

// Clean up and exit

return 0;

}

Creating a complete working example with all the details, including the encryption and file handling parts, is quite extensive. However, I can provide you with a basic outline of how you can structure the code for parallel file encryption using threads. You'll need to fill in the missing parts and adapt it to your specific needs.

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <openssl/aes.h>

#define NUM\_THREADS 6

#define BUFFER\_SIZE 1024

typedef struct {

char input\_file[BUFFER\_SIZE];

char output\_file[BUFFER\_SIZE];

char data\_str[4];

char hash[BUFFER\_SIZE];

} ThreadData;

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

void\* encrypt\_thread(void\* arg) {

ThreadData\* thread\_data = (ThreadData\*)arg;

// TODO: Perform AES encryption using thread\_data

// Lock the mutex before writing to the output file

pthread\_mutex\_lock(&mutex);

FILE\* out = fopen(thread\_data->output\_file, "a+");

if (out) {

fwrite(thread\_data->data\_str, 1, 4, out);

// TODO: Write encrypted data, hash, etc.

fclose(out);

}

// Unlock the mutex

pthread\_mutex\_unlock(&mutex);

return NULL;

}

int main() {

// Initialize variables and data structures

pthread\_t threads[NUM\_THREADS];

ThreadData thread\_data[NUM\_THREADS];

// Initialize mutex

if (pthread\_mutex\_init(&mutex, NULL) != 0) {

fprintf(stderr, "Mutex initialization failed\n");

return 1;

}

// Create threads for parallel encryption

for (int i = 0; i < NUM\_THREADS; ++i) {

// TODO: Set up thread\_data[i] with appropriate values

if (pthread\_create(&threads[i], NULL, encrypt\_thread, (void\*)&thread\_data[i]) != 0) {

fprintf(stderr, "Thread creation failed\n");

return 1;

}

}

// Wait for all threads to finish

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_join(threads[i], NULL);

}

// Clean up and exit

return 0;

}

```

Please note that you'll need to:

1. Implement the actual AES encryption logic using the `AES\_encrypt` function from OpenSSL.

2. Fill in the missing parts of the code, such as setting up the `ThreadData` structure and handling file I/O.

3. Adapt the code to your actual data structures and logic.

This code provides the basic structure for parallel file encryption using threads and mutexes. Remember to handle errors, manage memory properly, and thoroughly test your implementation to ensure its correctness and performance.

Certainly! Here's a modified version of the code, integrated with your provided code and labeled with `// TODO:` comments where you need to fill in your specific logic:

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <openssl/aes.h>

#define NUM\_THREADS 6

#define BUFFER\_SIZE 1024

typedef struct {

char input\_file[BUFFER\_SIZE];

char output\_file[BUFFER\_SIZE];

char data\_str[4];

char hash[BUFFER\_SIZE];

} ThreadData;

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

void\* encrypt\_thread(void\* arg) {

ThreadData\* thread\_data = (ThreadData\*)arg;

// TODO: Perform AES encryption using thread\_data

// e.g., Call your encryption function here

// Lock the mutex before writing to the output file

pthread\_mutex\_lock(&mutex);

FILE\* out = fopen(thread\_data->output\_file, "a+");

if (out) {

fwrite(thread\_data->data\_str, 1, 4, out);

// TODO: Write encrypted data, hash, etc.

// ...

fclose(out);

}

// Unlock the mutex

pthread\_mutex\_unlock(&mutex);

return NULL;

}

int main() {

// Initialize variables and data structures

pthread\_t threads[NUM\_THREADS];

ThreadData thread\_data[NUM\_THREADS];

// Initialize mutex

if (pthread\_mutex\_init(&mutex, NULL) != 0) {

fprintf(stderr, "Mutex initialization failed\n");

return 1;

}

// Create threads for parallel encryption

for (int i = 0; i < NUM\_THREADS; ++i) {

// TODO: Set up thread\_data[i] with appropriate values

// You need to set the input\_file, output\_file, data\_str, and hash fields

// based on your logic

// Example:

snprintf(thread\_data[i].input\_file, BUFFER\_SIZE, "input%d.txt", i);

snprintf(thread\_data[i].output\_file, BUFFER\_SIZE, "output%d.txt", i);

snprintf(thread\_data[i].data\_str, sizeof(thread\_data[i].data\_str), "data%d", i);

snprintf(thread\_data[i].hash, BUFFER\_SIZE, "hash%d", i);

if (pthread\_create(&threads[i], NULL, encrypt\_thread, (void\*)&thread\_data[i]) != 0) {

fprintf(stderr, "Thread creation failed\n");

return 1;

}

}

// Wait for all threads to finish

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_join(threads[i], NULL);

}

// Clean up and exit

return 0;

}

```

Replace the example values in the `TODO` sections with the appropriate logic from your original code. This modified version maintains the structure of parallelizing file encryption using threads and mutexes while incorporating your encryption and file handling logic.