To process multiple files concurrently (in batches of 5) for moving and encrypting, you'll need to implement a mechanism that allows you to iterate through the files in the source directory, select batches of files, and then move and encrypt them. You can achieve this using multi-threading or asynchronous programming. Here's a high-level outline of how you could modify your code to handle this:

1. Import necessary libraries for multi-threading or asynchronous programming.

2. Modify the `server` function or create a new function responsible for handling file moving and encryption in batches.

3. Use a loop to iterate through the files in the source directory.

4. For every batch of 5 files (or however many you want to process at a time), create threads or asynchronous tasks to move and encrypt each file.

5. Wait for all threads/tasks in the batch to complete before moving on to the next batch.

Here's a pseudo-code example using pthreads for multi-threading:

```c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

// Include other necessary headers...

#define NUM\_THREADS 5

#define NUM\_FILES\_PER\_BATCH 5

struct ThreadData {

char file\_paths[NUM\_FILES\_PER\_BATCH][BUFFER\_SIZE];

// Add any other necessary data for threads...

};

void \*process\_files(void \*thread\_arg) {

struct ThreadData \*data = (struct ThreadData \*)thread\_arg;

for (int i = 0; i < NUM\_FILES\_PER\_BATCH; i++) {

char \*file\_path = data->file\_paths[i];

// Move the file...

// Encrypt the file...

}

// Clean up and return...

}

int main() {

// Initialize SSL and other resources...

// Iterate through files and create batches for processing

while (/\* There are more files to process \*/) {

struct ThreadData thread\_data;

for (int i = 0; i < NUM\_FILES\_PER\_BATCH; i++) {

// Load file paths into thread\_data...

}

pthread\_t threads[NUM\_THREADS];

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

pthread\_create(&threads[i], NULL, process\_files, (void \*)&thread\_data);

}

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

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

}

}

// Clean up SSL and other resources...

return 0;

}

```

This is just a rough example, and you'll need to adapt it to your specific use case and environment. Be aware of thread safety when accessing shared resources and ensure proper error handling. If you prefer asynchronous programming, you could use libraries like libuv or libevent to achieve similar concurrency.

Sure, here's how you could modify your existing `server` function to handle file moving and encryption in batches of 5. This example uses multi-threading with pthreads to achieve concurrency:

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <openssl/ssl.h>

#include <openssl/sha.h>

#include <dirent.h>

#define BUFFER\_SIZE 1024

#define NUM\_THREADS 5

struct ThreadData {

char file\_paths[NUM\_THREADS][BUFFER\_SIZE];

};

void \*process\_files(void \*thread\_arg) {

struct ThreadData \*data = (struct ThreadData \*)thread\_arg;

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

char \*file\_path = data->file\_paths[i];

// Move the file...

// Encrypt the file...

}

// Clean up and return...

}

int server(SSL \*ssl, char \*file\_addr) {

// Accept client request...

struct ThreadData thread\_data;

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

// Load file paths into thread\_data...

}

pthread\_t threads[NUM\_THREADS];

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

pthread\_create(&threads[i], NULL, process\_files, (void \*)&thread\_data);

}

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

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

}

// Respond to client...

return 1;

}

```

In this example, the `server` function now initializes a `ThreadData` struct with file paths for each thread. It then creates and joins threads to process batches of files. The `process\_files` function is responsible for moving and encrypting the files within each thread.

Please note that you'll need to adapt this code to your specific use case, and you should handle any potential synchronization issues that might arise when multiple threads are accessing shared resources. Additionally, consider using a thread-safe data structure or synchronization mechanism to ensure proper communication and coordination between threads.

#include <stdio.h>

#include <string.h>

#include <openssl/ssl.h>

#include <openssl/sha.h>

#include <dirent.h>

#define BUFFER\_SIZE 1024

struct file\_info

{

char file\_name\_to\_client[BUFFER\_SIZE];

// other members here...

};

int file\_counter = 0;

void process\_files(SSL \*ssl, char \*file\_addr);

// ... other function declarations ...

int main()

{

// ... SSL initialization ...

char \*file\_addr = "/path/to/files";

process\_files(ssl, file\_addr);

// ... SSL cleanup ...

return 0;

}

void process\_files(SSL \*ssl, char \*file\_addr)

{

DIR \*folder\_ptr = opendir(file\_addr);

struct dirent \*fd\_ptr;

if (!folder\_ptr)

{

printf("Destination does not exist\n");

return;

}

struct file\_info info;

while ((fd\_ptr = readdir(folder\_ptr)) != NULL)

{

if (fd\_ptr->d\_type == DT\_REG && file\_counter < 5)

{

char path\_to\_file[BUFFER\_SIZE];

snprintf(path\_to\_file, sizeof(path\_to\_file), "%s/%s", file\_addr, fd\_ptr->d\_name);

// Assuming the gethash, compare hashes, move\_file functions are defined elsewhere

struct file\_info file\_info = gethash(path\_to\_file);

// Assuming the verifyhash function is defined elsewhere

if (verifyhash(file\_info.hash, file\_addr))

{

move\_file(path\_to\_file);

if (encrypt\_file(path\_to\_file, file\_info.hash))

{

strncpy(info.file\_name\_to\_client, fd\_ptr->d\_name, sizeof(info.file\_name\_to\_client));

file\_counter++;

}

else

{

secure\_remove(path\_to\_file);

}

}

else

{

printf("File verification failed!\n");

}

}

if (file\_counter >= 5)

{

// Perform encryption for the 5 files here...

// Reset the counter for the next batch

file\_counter = 0;

}

}

closedir(folder\_ptr);

}

// ... other function implementations ...

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <unistd.h>

#include <wait.h>

#define BUFFER\_SIZE 256

// ... Other code and definitions ...

int main(int argc, char \*argv[]) {

// Initialize configuration and variables

// Create an array to store batch of file paths

char \*file\_paths[5] = { NULL };

int file\_count = 0;

// Loop to process files in batches

while (1) {

// Perform file verification and get file\_path

char \*file\_path = /\* Get the file path \*/;

// Add the file\_path to the batch

file\_paths[file\_count] = strdup(file\_path);

file\_count++;

// When the batch is full, process it

if (file\_count == 5) {

// Process the batch

process\_batch(file\_paths);

// Clean up the batch array

for (int i = 0; i < 5; i++) {

free(file\_paths[i]);

file\_paths[i] = NULL;

}

file\_count = 0;

}

}

// Clean up any remaining files in the last batch

for (int i = 0; i < file\_count; i++) {

free(file\_paths[i]);

}

return 0;

}

void process\_batch(char \*file\_paths[]) {

for (int i = 0; i < 5; i++) {

if (file\_paths[i]) {

// Perform encryption on file\_paths[i]

encrypt\_file(file\_paths[i]);

}

}

}

int encrypt\_file(char \*source\_file) {

// Perform encryption logic here

// ... Your existing encryption code ...

return 1; // Return success

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <unistd.h>

#include <wait.h>

#define BUFFER\_SIZE 256

// ... Other code and definitions ...

void process\_batch(char \*file\_paths[], int file\_count) {

for (int i = 0; i < file\_count; i++) {

if (file\_paths[i]) {

// Perform encryption on file\_paths[i]

if (encrypt\_file(file\_paths[i]) == 1) {

printf("File encrypted successfully: %s\n", file\_paths[i]);

} else {

printf("File encryption failed: %s\n", file\_paths[i]);

}

free(file\_paths[i]);

}

}

}

int main(int argc, char \*argv[]) {

// Initialize configuration and variables

// Create an array to store batch of file paths

char \*file\_paths[5] = { NULL };

int file\_count = 0;

// Loop to process files in batches

while (1) {

// Perform file verification and get file\_path

char \*file\_path = /\* Get the file path \*/;

// Add the file\_path to the batch

file\_paths[file\_count] = strdup(file\_path);

file\_count++;

// When the batch is full, process it

if (file\_count == 5) {

// Process the batch

process\_batch(file\_paths, file\_count);

// Reset batch variables

for (int i = 0; i < 5; i++) {

file\_paths[i] = NULL;

}

file\_count = 0;

}

}

// Clean up any remaining files in the last batch

process\_batch(file\_paths, file\_count);

return 0;

}

int encrypt\_file(char \*source\_file) {

// Implement your encryption logic here

// Modify this part according to your encryption code

char fullpath\_out\_filename[BUFFER\_SIZE] = {0};

char org\_file\_name[BUFFER\_SIZE] = {0};

char ext[BUFFER\_SIZE] = {0};

int ret = 0;

// ... Your existing code ...

// Replace this with your actual encryption logic

// Sample code:

ret = perform\_actual\_encryption(source\_file);

return ret; // Return success or failure

}

#include "../server.h"

// ... Other code and definitions ...

int main(int argc, char \*argv[]) {

// Initialize configuration and variables

// Create an array to store batch of file paths

char \*file\_paths[5] = { NULL };

int file\_count = 0;

// Loop to process files in batches

while (1) {

// Perform file verification and get file\_path

char \*file\_path = /\* Get the file path \*/;

// Add the file\_path to the batch

file\_paths[file\_count] = strdup(file\_path);

file\_count++;

// When the batch is full, process it

if (file\_count == 5) {

// Process the batch

process\_batch(file\_paths);

// Clean up the batch array

for (int i = 0; i < 5; i++) {

free(file\_paths[i]);

file\_paths[i] = NULL;

}

file\_count = 0;

}

}

// Clean up any remaining files in the last batch

for (int i = 0; i < file\_count; i++) {

free(file\_paths[i]);

}

return 0;

}

void process\_batch(char \*file\_paths[]) {

for (int i = 0; i < 5; i++) {

if (file\_paths[i]) {

char \*hash = /\* Calculate hash for the file as a string \*/;

// Perform encryption on file\_paths[i] with the calculated hash

encrypt\_file(file\_paths[i], hash);

// Free the hash memory if needed

free(hash);

}

}

}

int encrypt\_file(char \*source\_file, char \*hash) {

// ... Your existing encryption code ...

return 1; // Return success

}

// ... Rest of the code ...