#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_THREADS 6

// Function that each thread will execute

void \*compute(void \*thread\_id) {

long tid = (long)thread\_id;

printf("Thread %ld started\n", tid);

// Simulate some computation

for (int i = 0; i < 100000000; ++i) {

double result = i \* 2.0; // Some computation

}

printf("Thread %ld finished\n", tid);

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS];

int rc;

long t;

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

rc = pthread\_create(&threads[t], NULL, compute, (void \*)t);

if (rc) {

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(-1);

}

}

// Wait for all threads to finish

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

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

}

printf("All threads have finished\n");

return 0;

}

Creating a program that explicitly assigns threads to specific cores is a more complex task and might require platform-specific code. Below is an example using the `pthread` library to create six threads and manually bind each thread to a specific core. Note that thread-to-core binding is not guaranteed and might be affected by the operating system's scheduler.

```c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_THREADS 6

// Function that each thread will execute

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

long tid = (long)thread\_id;

printf("Thread %ld started\n", tid);

// Perform some work

// ...

printf("Thread %ld finished\n", tid);

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS];

int rc;

long t;

// Define CPU core assignments for each thread

int core\_assignments[NUM\_THREADS] = {0, 1, 2, 3, 4, 5}; // Assign threads to cores 0 through 5

// Set CPU core affinity for each thread

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

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

cpu\_set\_t cpuset;

CPU\_ZERO(&cpuset);

CPU\_SET(core\_assignments[t], &cpuset);

pthread\_attr\_setaffinity\_np(&attr, sizeof(cpu\_set\_t), &cpuset);

rc = pthread\_create(&threads[t], &attr, thread\_function, (void \*)t);

if (rc) {

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(-1);

}

}

// Wait for all threads to finish

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

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

}

printf("All threads have finished\n");

return 0;

}

```

This program demonstrates how to manually assign threads to specific cores using the `pthread\_attr\_setaffinity\_np` function. However, please note that thread affinity and core allocation can vary depending on the operating system and system load. Additionally, this example assumes that you have at least 6 cores available on your system.

To compile the program, use a command like:

```sh

gcc -o core\_binding\_program core\_binding\_program.c -lpthread

```

Remember that CPU core binding and affinity can be sensitive to system configurations and might require additional privileges on some systems. Make sure to test and adapt the code according to your specific environment.

Certainly! Here's an example of an encrypt function using a simple XOR encryption technique to perform some work in each thread:

```c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_THREADS 6

// Function that each thread will execute

void \*encrypt(void \*thread\_id) {

long tid = (long)thread\_id;

printf("Thread %ld started\n", tid);

// Plaintext to be encrypted

char plaintext[] = "This is a secret message.";

size\_t len = sizeof(plaintext) - 1; // Exclude the null terminator

// Encryption key

char key = 'X' + tid;

// Encrypt the plaintext using XOR

for (size\_t i = 0; i < len; ++i) {

plaintext[i] ^= key;

}

printf("Thread %ld encrypted: %s\n", tid, plaintext);

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS];

int rc;

long t;

// Define CPU core assignments for each thread

int core\_assignments[NUM\_THREADS] = {0, 1, 2, 3, 4, 5}; // Assign threads to cores 0 through 5

// Set CPU core affinity for each thread

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

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

cpu\_set\_t cpuset;

CPU\_ZERO(&cpuset);

CPU\_SET(core\_assignments[t], &cpuset);

pthread\_attr\_setaffinity\_np(&attr, sizeof(cpu\_set\_t), &cpuset);

rc = pthread\_create(&threads[t], &attr, encrypt, (void \*)t);

if (rc) {

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(-1);

}

}

// Wait for all threads to finish

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

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

}

printf("All threads have finished\n");

return 0;

}

```

This example replaces the "Perform some work" comment with an encryption task using XOR encryption. Each thread encrypts a predefined plaintext message using a different key based on its thread ID. Please note that XOR encryption is not secure for actual encryption purposes; it's used here for demonstration purposes only.

Certainly! Here's how you can modify your existing code to integrate the `verifyhash` function with multi-threading. I'll add the necessary parts to your existing code to create and manage threads for the `verifyhash` function, with each thread working on a different subset of files and running on a different core.

Please note that this is a basic outline, and you might need to adjust and refine it according to your specific requirements and the structure of your code.

```c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <openssl/ssl.h>

#include <dirent.h>

#define NUM\_THREADS 6

// Define constants and structures here

#define BUFFER\_SIZE 1024

struct ThreadData {

char file\_addr[BUFFER\_SIZE];

char file\_hash[BUFFER\_SIZE];

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

// Add any other data that you need to pass to the thread

};

// Function that each thread will execute

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

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

DIR \*folder\_ptr = opendir(thread\_data->file\_addr);

int ret\_val = 0;

if (!folder\_ptr) {

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

return NULL;

} else {

struct dirent \*fd\_ptr;

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

if (strcmp(fd\_ptr->d\_name, ".") == 0 || strcmp(fd\_ptr->d\_name, "..") == 0) {

continue;

} else if (fd\_ptr->d\_type == DT\_REG) {

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

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

// Get hash and check for verification

// ...

if (strcmp(info.hash, thread\_data->file\_hash) == 0) {

if (encrypt\_file(path\_to\_file, info.hash)) {

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

ret\_val = 1;

} else {

if (!secure\_remove(path\_to\_file)) {

remove(path\_to\_file);

}

}

break;

} else {

if (!secure\_remove(path\_to\_file)) {

remove(path\_to\_file);

}

}

}

}

closedir(folder\_ptr);

}

return (void \*)(intptr\_t)ret\_val;

}

int main() {

pthread\_t threads[NUM\_THREADS];

int rc;

// Initialize SSL library and other resources

// ...

// Define thread data for each thread

struct ThreadData thread\_data[NUM\_THREADS];

// Set data for each thread

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

snprintf(thread\_data[t].file\_addr, sizeof(thread\_data[t].file\_addr), "path\_to\_files\_%d", t);

snprintf(thread\_data[t].file\_hash, sizeof(thread\_data[t].file\_hash), "hash\_%d", t);

memset(thread\_data[t].file\_name\_to\_client, 0, sizeof(thread\_data[t].file\_name\_to\_client));

}

// Create and start threads

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

rc = pthread\_create(&threads[t], NULL, thread\_function, (void \*)&thread\_data[t]);

if (rc) {

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(EXIT\_FAILURE);

}

}

// Wait for all threads to finish

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

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

}

// Clean up resources

// ...

printf("All threads have finished\n");

return 0;

}

```

Please integrate this code with your existing codebase and make any necessary adjustments. This outline demonstrates how to create and manage threads for the `verifyhash` function with each thread working on a different subset of files. Remember to handle synchronization, resource cleanup, and any other necessary tasks specific to your application.