Certainly, I understand that you want to modify the provided code to utilize 6 CPU cores equally for parallel encryption of files. Here's the modified code that ensures the encryption process utilizes 6 CPU cores evenly:

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <pthread.h>

#include <openssl/evp.h>

#define BUF\_SIZE 1024

#define NUM\_THREADS 6 // Number of CPU cores to utilize

void handleErrors(void) {

printf("Error occurred\n");

exit(1);

}

void\* encryptFile(void\* arg) {

// ... (Same code as before)

}

int main() {

const char \*inputFolder = "input\_folder/";

DIR \*dir;

struct dirent \*ent;

if ((dir = opendir(inputFolder)) != NULL) {

pthread\_t threads[NUM\_THREADS];

// Create an array to track the thread index for each file

int threadIndices[NUM\_THREADS];

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

threadIndices[i] = i;

}

while ((ent = readdir(dir)) != NULL) {

if (ent->d\_type == DT\_REG) { // Regular file

// Determine which thread to use based on round-robin approach

int selectedThread = threadIndices[threadIndex % NUM\_THREADS];

pthread\_create(&threads[selectedThread], NULL, encryptFile, ent->d\_name);

threadIndex++;

}

}

// Wait for all threads to finish

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

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

}

closedir(dir);

} else {

perror("Error opening directory");

return 1;

}

printf("All files encrypted successfully!\n");

return 0;

}

```

In this modified code, we've made the following changes:

1. Defined `NUM\_THREADS` as the number of CPU cores to utilize (6 in your case).

2. Created an array `threadIndices` to keep track of which thread index each file should be assigned to, using a round-robin approach.

3. Inside the loop that processes files, we determine the selected thread based on the round-robin approach using `threadIndices[threadIndex % NUM\_THREADS]`.

4. After processing all files, we wait for all threads to finish using a loop that joins all threads.

Please make sure to replace `"input\_folder/"` with the appropriate path to your input folder. Also, replace the encryption key, if necessary. Compile the code and run the executable as previously explained. This code modification should ensure that the encryption process utilizes 6 CPU cores evenly for parallel processing.

~~~~~~

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <pthread.h>

#include <openssl/evp.h>

#define BUF\_SIZE 1024

#define NUM\_THREADS 6

void handleErrors(void) {

printf("Error occurred\n");

exit(1);

}

void\* encryptFile(void\* arg) {

// ... (Same code as before)

}

int main() {

const char \*inputFolder = "input\_folder/";

DIR \*dir;

struct dirent \*ent;

if ((dir = opendir(inputFolder)) != NULL) {

pthread\_t threads[NUM\_THREADS];

int threadIndex = 0;

while ((ent = readdir(dir)) != NULL && threadIndex < NUM\_THREADS) {

if (ent->d\_type == DT\_REG) { // Regular file

pthread\_create(&threads[threadIndex], NULL, encryptFile, ent->d\_name);

threadIndex++;

}

}

// Wait for all threads to finish

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

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

}

closedir(dir);

} else {

perror("Error opening directory");

return 1;

}

printf("All files encrypted successfully!\n");

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <pthread.h>

#include <openssl/evp.h>

#define BUF\_SIZE 1024

void handleErrors(void) {

fprintf(stderr, "Error occurred\n");

exit(1);

}

void\* encryptFile(void\* arg) {

const char \*key = "your\_encryption\_key"; // Change this to a secure method of key storage

const char \*inputFolder = "input\_folder/";

const char \*outputFolder = "output\_folder/";

const char \*filename = (const char \*)arg;

char inputFile[256];

char outputFile[256];

snprintf(inputFile, sizeof(inputFile), "%s%s", inputFolder, filename);

snprintf(outputFile, sizeof(outputFile), "%s%s.enc", outputFolder, filename);

FILE \*inFile = fopen(inputFile, "rb");

FILE \*outFile = fopen(outputFile, "wb");

if (!inFile || !outFile) {

perror("Error opening files");

return NULL;

}

const int keyLength = strlen(key);

EVP\_CIPHER\_CTX \*ctx;

ctx = EVP\_CIPHER\_CTX\_new();

if (!ctx) handleErrors();

if (1 != EVP\_EncryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), NULL, (unsigned char \*)key, NULL))

handleErrors();

unsigned char bufferIn[BUF\_SIZE];

unsigned char bufferOut[BUF\_SIZE + EVP\_MAX\_BLOCK\_LENGTH];

int bytesRead;

int bytesWritten;

while ((bytesRead = fread(bufferIn, 1, BUF\_SIZE, inFile)) > 0) {

if (1 != EVP\_EncryptUpdate(ctx, bufferOut, &bytesWritten, bufferIn, bytesRead))

handleErrors();

fwrite(bufferOut, 1, bytesWritten, outFile);

}

if (1 != EVP\_EncryptFinal\_ex(ctx, bufferOut, &bytesWritten))

handleErrors();

fwrite(bufferOut, 1, bytesWritten, outFile);

EVP\_CIPHER\_CTX\_free(ctx);

fclose(inFile);

fclose(outFile);

printf("File %s encrypted successfully!\n", filename);

return NULL;

}

int main() {

const char \*inputFolder = "input\_folder/";

DIR \*dir;

struct dirent \*ent;

if ((dir = opendir(inputFolder)) != NULL) {

pthread\_t threads[6]; // Using 6 threads for 6 cores

int threadIndex = 0;

while ((ent = readdir(dir)) != NULL) {

if (ent->d\_type == DT\_REG) { // Regular file

pthread\_create(&threads[threadIndex], NULL, encryptFile, ent->d\_name);

threadIndex++;

if (threadIndex >= 6) { // Using 6 threads for 6 cores

for (int i = 0; i < 6; i++) { // Using 6 threads for 6 cores

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

}

threadIndex = 0;

}

}

}

// Wait for remaining threads to finish

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

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

}

closedir(dir);

} else {

perror("Error opening directory");

return 1;

}

printf("All files encrypted successfully!\n");

return 0;

}

Of course! I've integrated the code you provided with the information from our previous conversation about allocating each thread to a specific core for parallel encryption of files. Please note that while the following code sets affinity to specific cores, it's important to remember that thread affinity can be a complex topic and actual behavior might vary depending on your system and OS scheduler.

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <pthread.h>

#include <openssl/evp.h>

#define BUF\_SIZE 1024

void handleErrors(void) {

fprintf(stderr, "Error occurred\n");

exit(1);

}

void\* encryptFile(void\* arg) {

const char \*key = "your\_encryption\_key"; // Change this to a secure method of key storage

const char \*inputFolder = "input\_folder/";

const char \*outputFolder = "output\_folder/";

const char \*filename = (const char \*)arg;

char inputFile[256];

char outputFile[256];

snprintf(inputFile, sizeof(inputFile), "%s%s", inputFolder, filename);

snprintf(outputFile, sizeof(outputFile), "%s%s.enc", outputFolder, filename);

FILE \*inFile = fopen(inputFile, "rb");

FILE \*outFile = fopen(outputFile, "wb");

if (!inFile || !outFile) {

perror("Error opening files");

return NULL;

}

const int keyLength = strlen(key);

EVP\_CIPHER\_CTX \*ctx;

ctx = EVP\_CIPHER\_CTX\_new();

if (!ctx) handleErrors();

if (1 != EVP\_EncryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), NULL, (unsigned char \*)key, NULL))

handleErrors();

unsigned char bufferIn[BUF\_SIZE];

unsigned char bufferOut[BUF\_SIZE + EVP\_MAX\_BLOCK\_LENGTH];

int bytesRead;

int bytesWritten;

while ((bytesRead = fread(bufferIn, 1, BUF\_SIZE, inFile)) > 0) {

if (1 != EVP\_EncryptUpdate(ctx, bufferOut, &bytesWritten, bufferIn, bytesRead))

handleErrors();

fwrite(bufferOut, 1, bytesWritten, outFile);

}

if (1 != EVP\_EncryptFinal\_ex(ctx, bufferOut, &bytesWritten))

handleErrors();

fwrite(bufferOut, 1, bytesWritten, outFile);

EVP\_CIPHER\_CTX\_free(ctx);

fclose(inFile);

fclose(outFile);

printf("File %s encrypted successfully!\n", filename);

return NULL;

}

int main() {

const char \*inputFolder = "input\_folder/";

DIR \*dir;

struct dirent \*ent;

if ((dir = opendir(inputFolder)) != NULL) {

pthread\_t threads[6]; // Using 6 threads for 6 cores

int threadIndex = 0;

cpu\_set\_t cpuset;

while ((ent = readdir(dir)) != NULL) {

if (ent->d\_type == DT\_REG) { // Regular file

int thread\_id = threadIndex % 6;

CPU\_ZERO(&cpuset);

CPU\_SET(thread\_id, &cpuset);

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

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

pthread\_create(&threads[threadIndex], &attr, encryptFile, ent->d\_name);

threadIndex++;

if (threadIndex >= 6) { // Using 6 threads for 6 cores

for (int i = 0; i < 6; i++) { // Using 6 threads for 6 cores

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

}

threadIndex = 0;

}

}

}

// Wait for remaining threads to finish

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

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

}

closedir(dir);

} else {

perror("Error opening directory");

return 1;

}

printf("All files encrypted successfully!\n");

return 0;

}

```

Please remember that this code may not behave exactly as expected on all systems, as thread affinity and core allocation can be influenced by various factors including the operating system's scheduler and CPU architecture.