To achieve the multithreading file transfer function and allocate 10 cores for file processing and encryption, you can use the pthreads library in C. Below is a sample program that demonstrates how to do this. Please note that this example assumes you have a function `encryptFile()` to handle the encryption process.

```c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_CORES 12

#define NUM\_THREADS 10

#define NUM\_FILES 100 // Total number of files to transfer and encrypt

void\* transferAndEncrypt(void\* arg) {

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

// Simulate file transfer - replace this with actual file transfer logic

printf("Transferring file: %s\n", filename);

// Simulate encryption - replace this with actual encryption logic

printf("Encrypting file: %s\n", filename);

encryptFile(filename);

// Simulate processing on core

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

// Some CPU-intensive operation

}

printf("Completed processing file: %s\n", filename);

return NULL;

}

int main() {

pthread\_t threads[NUM\_THREADS];

char filenames[NUM\_FILES][50]; // Assuming filenames have a maximum length of 50 characters

// Populate the filenames (replace this with actual file names)

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

snprintf(filenames[i], sizeof(filenames[i]), "file%d.txt", i);

}

// Create threads for file transfer and encryption

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

int fileIndex = i \* (NUM\_FILES / NUM\_THREADS); // Distribute files among threads

pthread\_create(&threads[i], NULL, transferAndEncrypt, (void\*)filenames[fileIndex]);

}

// Wait for all threads to complete

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

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

}

printf("All files processed.\n");

return 0;

}

```

In this example, we use 10 threads to transfer and encrypt files concurrently, and each thread is assigned a portion of the total files to process. The `encryptFile()` function represents the actual encryption logic for a single file, and you should implement this function to perform the encryption using an appropriate encryption algorithm.

Please note that the file transfer and encryption parts are simulated in this example. In a real-world scenario, you would need to replace the file transfer logic with actual file I/O operations, and implement a proper encryption function based on your security requirements.

Also, keep in mind that CPU core allocation is typically managed by the operating system, and this code doesn't explicitly allocate cores but rather utilizes multiple threads, which the OS may distribute across the available cores. If you have specific requirements to allocate cores, you might need to use platform-specific functions or APIs.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

int main() {

// Create a list of files to transfer

char \*files[] = {"file1.txt", "file2.txt", "file3.txt", "file4.txt", "file5.txt", "file6.txt", "file7.txt", "file8.txt", "file9.txt", "file10.txt"};

// Create a thread pool with 10 threads

pthread\_t threads[10];

// Iterate over the files and start a thread for each file

for (int i = 0; i < 10; i++) {

pthread\_create(&threads[i], NULL, transfer\_file, (void \*)files[i]);

}

// Wait for all threads to finish

for (int i = 0; i < 10; i++) {

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

}

// Encrypt the files

encrypt\_files();

// Create a new thread pool with 10 threads

pthread\_t threads2[10];

// Iterate over the files and start a thread for each file

for (int i = 0; i < 10; i++) {

pthread\_create(&threads2[i], NULL, transfer\_file, (void \*)files[i]);

}

// Wait for all threads to finish

for (int i = 0; i < 10; i++) {

pthread\_join(threads2[i], NULL);

}

return 0;

}

void \*transfer\_file(void \*file) {

// Get the file name

char \*filename = (char \*)file;

// Start the file transfer

printf("Transferring file %s\n", filename);

sleep(1);

// Notify the main thread that the file transfer is complete

printf("File transfer complete\n");

return NULL;

}

void encrypt\_files() {

// Encrypt the files

printf("Encrypting files\n");

sleep(1);

// Notify the main thread that the encryption is complete

printf("Encryption complete\n");

}

Use code with caution. Learn more

This program will create a list of 10 files and then start a thread for each file. The threads will transfer the files to a different location. Once the files have been transferred, the program will encrypt the files and then start a new thread for each file. The threads will transfer the encrypted files to a different location.

The program will use 10 cores out of 12 to transfer the files. The first 10 cores will be used to transfer the files, and the remaining 2 cores will be used to encrypt the files.

To run the program, you will need to have the pthread library installed on your system. You can find more information about the pthread library here: https://computing.llnl.gov/tutorials/pthreads/

Once you have the pthread library installed, you can compile and run the program by typing the following commands into a terminal:

gcc -o file\_transfer file\_transfer.c -pthread

./file\_transfer

This will start the file transfer and encryption process.

Sure, I'll modify the example to use the `file\_handle` function for file transfer and the `encrypt\_file` function for file encryption. Let's assume that the `file\_handle` function takes the source filename and destination filename as arguments, and the `encrypt\_file` function takes the filename to be encrypted.

```c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_CORES 12

#define NUM\_THREADS 10

#define NUM\_FILES 100

// Placeholder function for file transfer

void file\_handle(const char\* src\_filename, const char\* dest\_filename) {

// Simulated file transfer process

printf("Transferring file: %s to %s\n", src\_filename, dest\_filename);

// Your actual file transfer code here

}

// Placeholder function for file encryption

void encrypt\_file(const char\* filename) {

// Simulated encryption process

printf("Encrypting file: %s\n", filename);

// Your actual encryption code here

}

// Function to transfer and encrypt files

void\* transferAndEncryptFiles(void\* arg) {

char\*\* filenames = (char\*\*)arg;

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

// Get the file name to transfer and encrypt

const char\* src\_filename = filenames[i];

const char\* dest\_filename = "destination\_directory/"; // Set your destination directory

char dest\_filepath[256];

snprintf(dest\_filepath, sizeof(dest\_filepath), "%sencrypted\_%s", dest\_filename, src\_filename);

// Transfer the file

file\_handle(src\_filename, dest\_filepath);

// Encrypt the file

encrypt\_file(dest\_filepath);

}

return NULL;

}

int main() {

// Assuming you have an array of file names (replace these with actual file paths)

char\* fileNames[NUM\_FILES] = {

"file1.txt", "file2.txt", "file3.txt", /\* Add more files here \*/

};

// Divide files among 10 threads

int filesPerThread = NUM\_FILES / NUM\_THREADS;

pthread\_t threads[NUM\_THREADS];

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

// Create thread and pass the array of file names to each thread

pthread\_create(&threads[i], NULL, transferAndEncryptFiles, &fileNames[i \* filesPerThread]);

}

// Wait for all threads to complete

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

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

}

// Now the files are transferred and encrypted.

return 0;

}

```

In this updated example, we use the `file\_handle` function to transfer the file from the source directory to the destination directory, and then we use the `encrypt\_file` function to encrypt the transferred file in the destination directory. The `transferAndEncryptFiles` function handles both the file transfer and encryption for each file, and we divide the files among the 10 threads to perform these operations in parallel.

Please make sure to replace `"destination\_directory/"` in the code with your actual destination directory for the encrypted files. Also, implement the actual file transfer and encryption logic in the `file\_handle` and `encrypt\_file` functions accordingly.

Sure, I'll modify the code to take file paths from a configuration file and use those paths for file transfer and encryption. For this example, we'll assume that the configuration file is a simple text file with two lines: the first line contains the source directory path, and the second line contains the destination directory path.

Let's assume the configuration file is named "config.txt" and has the following content:

```

/source\_directory/

/destination\_directory/

```

Here's the modified code:

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#define NUM\_CORES 12

#define NUM\_THREADS 10

#define NUM\_FILES 100

#define MAX\_PATH\_LENGTH 256

// Placeholder function for file transfer

void file\_handle(const char\* src\_filename, const char\* dest\_filename) {

// Simulated file transfer process

printf("Transferring file: %s to %s\n", src\_filename, dest\_filename);

// Your actual file transfer code here

}

// Placeholder function for file encryption

void encrypt\_file(const char\* filename) {

// Simulated encryption process

printf("Encrypting file: %s\n", filename);

// Your actual encryption code here

}

// Function to transfer and encrypt files

void\* transferAndEncryptFiles(void\* arg) {

char\*\* filenames = (char\*\*)arg;

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

// Get the file name to transfer and encrypt

const char\* src\_filename = filenames[i];

const char\* dest\_filename = filenames[i + NUM\_FILES];

// Transfer the file

file\_handle(src\_filename, dest\_filename);

// Encrypt the file

encrypt\_file(dest\_filename);

}

return NULL;

}

int main() {

char source\_dir[MAX\_PATH\_LENGTH];

char destination\_dir[MAX\_PATH\_LENGTH];

// Read the configuration from the file

FILE\* config\_file = fopen("config.txt", "r");

if (config\_file == NULL) {

printf("Error: Could not open the config file.\n");

return 1;

}

// Read the source and destination directory paths from the config file

if (fgets(source\_dir, sizeof(source\_dir), config\_file) == NULL ||

fgets(destination\_dir, sizeof(destination\_dir), config\_file) == NULL) {

printf("Error: Failed to read from the config file.\n");

fclose(config\_file);

return 1;

}

// Remove newline characters from the directory paths

source\_dir[strcspn(source\_dir, "\n")] = '\0';

destination\_dir[strcspn(destination\_dir, "\n")] = '\0';

fclose(config\_file);

// Assuming you have an array of file names (replace these with actual file paths)

char fileNames[NUM\_FILES \* 2][MAX\_PATH\_LENGTH] = {

/\* Add source file paths here \*/

/\* Add destination file paths here \*/

};

// Combine source and destination directory paths with file names

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

snprintf(fileNames[i], sizeof(fileNames[i]), "%s%s", source\_dir, "file1.txt"); // Replace "file1.txt" with actual file names

snprintf(fileNames[i + NUM\_FILES], sizeof(fileNames[i + NUM\_FILES]), "%s%s%s", destination\_dir, "/encrypted\_", "file1.txt");

}

// Divide files among 10 threads

int filesPerThread = NUM\_FILES / NUM\_THREADS;

pthread\_t threads[NUM\_THREADS];

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

// Create thread and pass the array of file names to each thread

pthread\_create(&threads[i], NULL, transferAndEncryptFiles, &fileNames[i \* filesPerThread]);

}

// Wait for all threads to complete

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

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

}

// Now the files are transferred and encrypted.

return 0;

}

```

In this modified code, we read the source and destination directory paths from the "config.txt" file and then use them to create the full file paths for the source and destination files in the `fileNames` array. The `transferAndEncryptFiles` function handles both the file transfer and encryption for each file, and we divide the files among the 10 threads to perform these operations in parallel.

Please ensure that the paths in the configuration file are correct and that the file names in the `fileNames` array are replaced with actual file names accordingly. Additionally, implement the actual file transfer and encryption logic in the `file\_handle` and `encrypt\_file` functions as per your requirements.

Sure, I'll update the example to read the file paths from a configuration file, transfer the files from the source directory to the destination directory using the `file\_handle` function, and then encrypt them using the `encrypt\_file` function.

For simplicity, I'll use a placeholder function called `read\_config` to read the file paths from the configuration file. You should replace this function with your actual configuration file reading mechanism.

```c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_CORES 12

#define NUM\_THREADS 10

#define NUM\_FILES 100

// Placeholder function for file transfer

void file\_handle(const char\* src\_filename, const char\* dest\_filename) {

// Simulated file transfer process

printf("Transferring file: %s to %s\n", src\_filename, dest\_filename);

// Your actual file transfer code here

}

// Placeholder function for file encryption

void encrypt\_file(const char\* filename) {

// Simulated encryption process

printf("Encrypting file: %s\n", filename);

// Your actual encryption code here

}

// Placeholder function to read file paths from the configuration file

void read\_config(char\* fileNames[NUM\_FILES]) {

// Simulated reading of file paths from the configuration file

// Replace this with your actual configuration file reading code

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

snprintf(fileNames[i], 256, "source\_directory/file%d.txt", i + 1);

}

}

// Function to transfer and encrypt files

void\* transferAndEncryptFiles(void\* arg) {

char\*\* filenames = (char\*\*)arg;

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

// Get the file name to transfer and encrypt

const char\* src\_filename = filenames[i];

const char\* dest\_filename = "destination\_directory/"; // Set your destination directory

char dest\_filepath[256];

snprintf(dest\_filepath, sizeof(dest\_filepath), "%sencrypted\_%s", dest\_filename, src\_filename);

// Transfer the file

file\_handle(src\_filename, dest\_filepath);

// Encrypt the file

encrypt\_file(dest\_filepath);

}

return NULL;

}

int main() {

// Assuming you have an array of file names read from the configuration file

char\* fileNames[NUM\_FILES];

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

fileNames[i] = (char\*)malloc(256 \* sizeof(char));

}

// Read file paths from the configuration file

read\_config(fileNames);

// Divide files among 10 threads

int filesPerThread = NUM\_FILES / NUM\_THREADS;

pthread\_t threads[NUM\_THREADS];

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

// Create thread and pass the array of file names to each thread

pthread\_create(&threads[i], NULL, transferAndEncryptFiles, &fileNames[i \* filesPerThread]);

}

// Wait for all threads to complete

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

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

}

// Free memory allocated for filenames

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

free(fileNames[i]);

}

return 0;

}

```

In this updated example, we use the `read\_config` function to read the file paths from the configuration file (replace this with your actual configuration file reading logic). Then, the `transferAndEncryptFiles` function handles both the file transfer and encryption for each file, and we divide the files among the 10 threads to perform these operations in parallel.

Please ensure that you modify the configuration file reading logic (`read\_config`) to correctly read the file paths from your actual configuration file. Also, implement the actual file transfer and encryption logic in the `file\_handle` and `encrypt\_file` functions accordingly.