It looks like the issue you're encountering might be related to resource limitations, such as file descriptor limits, that are preventing you from opening more than 500 files. To handle files continuously without hitting these limits, you could consider modifying your code as follows:

1. \*\*Close Files\*\*: Make sure that after processing each file, you close both the input and output files. Open files consume file descriptors, and not closing them can lead to running out of available file descriptors.

2. \*\*Thread Management\*\*: You're creating threads for each file, which can lead to resource exhaustion. Instead of creating a new thread for each file, consider using a thread pool to limit the number of active threads.

Here's a modified version of your code that addresses these issues:

```c

// ... (Header includes and definitions)

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

int activeThreads = 0;

void\* encryptFile(void\* arg) {

// ... (Same as before)

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

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

if (!inFile || !outFile) {

perror("Error opening files");

return NULL;

}

// ... (Encryption and execve code)

fclose(inFile);

fclose(outFile);

pthread\_mutex\_lock(&mutex);

activeThreads--;

pthread\_mutex\_unlock(&mutex);

return NULL;

}

int main() {

// ... (Folder checks)

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

pthread\_t threads[MAX\_THREADS];

int threadIndex = 0;

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

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

pthread\_mutex\_lock(&mutex);

if (activeThreads >= MAX\_THREADS) {

pthread\_mutex\_unlock(&mutex);

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

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

}

threadIndex = 0;

} else {

pthread\_mutex\_unlock(&mutex);

}

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

threadIndex++;

pthread\_mutex\_lock(&mutex);

activeThreads++;

pthread\_mutex\_unlock(&mutex);

}

}

// 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;

}

```

In this version, a mutex (`pthread\_mutex\_t`) is used to control access to the `activeThreads` variable to ensure thread-safe modifications. The active thread count is used to limit the number of concurrently running encryption threads. This should help you handle files continuously without hitting resource limits.

To continuously process files as they're added to the folder, you can implement a loop that keeps scanning the folder for new files. Here's a modified version of your code that adds this functionality:

```c

// ... (Header includes and definitions)

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

int activeThreads = 0;

void\* encryptFile(void\* arg) {

// ... (Same as before)

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

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

if (!inFile || !outFile) {

perror("Error opening files");

return NULL;

}

// ... (Encryption and execve code)

fclose(inFile);

fclose(outFile);

pthread\_mutex\_lock(&mutex);

activeThreads--;

pthread\_mutex\_unlock(&mutex);

return NULL;

}

void\* fileWatcher(void\* arg) {

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

while (1) {

DIR \*dir = opendir(inputFolder);

if (dir) {

struct dirent \*ent;

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

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

pthread\_mutex\_lock(&mutex);

if (activeThreads < MAX\_THREADS) {

pthread\_t thread;

pthread\_create(&thread, NULL, encryptFile, ent->d\_name);

activeThreads++;

}

pthread\_mutex\_unlock(&mutex);

}

}

closedir(dir);

} else {

perror("Error opening directory");

}

sleep(1); // Sleep for some time before scanning the folder again

}

return NULL;

}

int main() {

// ... (Folder checks)

pthread\_t watcherThread;

pthread\_create(&watcherThread, NULL, fileWatcher, NULL);

// This main thread will wait forever as the watcherThread keeps running

pthread\_join(watcherThread, NULL);

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

return 0;

}

```

In this version, a new thread `fileWatcher` is created to continuously scan the input folder for new files. If a new file is found and the maximum number of active threads hasn't been reached, a new encryption thread is created to handle the new file. The `fileWatcher` thread runs in an infinite loop, repeatedly scanning the folder and starting encryption threads as needed.

The main thread simply waits for the `fileWatcher` thread to complete, which will never happen since the `fileWatcher` thread runs indefinitely. This way, you can ensure that files are encrypted continuously as they are added to the input folder.

I'll incorporate thread management into your code snippet to handle multiple files concurrently. However, please note that your original code is a bit fragmented, and certain custom functions are not defined within the provided context. I'll attempt to integrate the threading logic into your code as best as I can. You should adapt this to your actual implementation and add the missing pieces accordingly.

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

#include <time.h>

#define MAX\_THREADS 6

#define BUFFER\_SIZE 1024

// ... (Struct and union definitions)

// Define your custom functions here

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

// Define other custom functions used in your code

void\* processFile(void\* arg) {

char\* source\_file = (char\*)arg;

char hash[BUFFER\_SIZE] = "example\_hash"; // Replace with your hash generation code

int ret = encrypt\_file(source\_file, hash); // Call your encryption function here

if (ret == 0) {

printf("Encryption failed for file %s\n", source\_file);

} else {

printf("Encryption succeeded for file %s\n", source\_file);

}

return NULL;

}

int main() {

// Assuming you have an array of source file paths

char\* source\_files[] = {

"file1.txt",

"file2.txt",

// ... (add more files)

};

int num\_files = sizeof(source\_files) / sizeof(source\_files[0]);

pthread\_t threads[MAX\_THREADS];

int threadIndex = 0;

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

if (threadIndex >= MAX\_THREADS) {

for (int j = 0; j < MAX\_THREADS; j++) {

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

}

threadIndex = 0;

}

pthread\_create(&threads[threadIndex], NULL, processFile, source\_files[i]);

threadIndex++;

}

// Wait for remaining threads to finish

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

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

}

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

return 0;

}

```

Again, keep in mind that you'll need to replace `"example\_hash"` with your hash generation logic and ensure that your custom functions are correctly implemented. This modified code integrates the thread management logic into your existing code structure.

If you want to process all source files from a directory and encrypt them using multiple threads, you can use the `opendir` and `readdir` functions to iterate through the files in the directory. Here's how you could modify the code to achieve that:

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <dirent.h> // Include the dirent library for directory operations

#include <pthread.h> // Include the pthread library

// Define constants and structures used in the code

#define BUFFER\_SIZE 1024 // Example buffer size, change as needed

typedef unsigned int UnsignedInt;

typedef unsigned char UnsignedChar;

union RandomPrefix {

UnsignedInt number;

UnsignedChar str[4];

};

union TimeData {

time\_t time\_num;

char time\_char[8];

};

UnsignedChar key[33];

UnsignedChar iv[17];

// Define other functions used in the code

int aes\_encrypt(const char \*input\_file, const char \*output\_file, char \*str, char \*hash) {

// The content of the aes\_encrypt function

// ...

return 1;

}

void encrypt\_data(char \*filename, char \*iv\_seed, char \*buffer) {

// Implementation of the encrypt\_data function

// ...

}

void secure\_remove(const char \*filename) {

// Implementation of the secure\_remove function

// ...

}

int custom\_strlen(const char \*str) {

// Implementation of the custom\_strlen function

// ...

}

void secure\_free(void \*ptr, size\_t size) {

// Implementation of the secure\_free function

// ...

}

// Function to be run by each thread

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

// Cast the argument to the appropriate type

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

// Perform the encryption for this thread

// ...

// Clean up and exit the thread

pthread\_exit(NULL);

}

int main() {

// Initialize threading variables

pthread\_t threads[6]; // Array to store thread IDs

// Directory containing source files

const char \*source\_directory = "/path/to/source/directory";

// Open the directory

DIR \*dir = opendir(source\_directory);

if (dir == NULL) {

perror("opendir");

return 1;

}

// Read files from the directory

struct dirent \*entry;

int thread\_count = 0;

while ((entry = readdir(dir)) != NULL && thread\_count < 6) {

// Skip "." and ".." entries and directories

if (entry->d\_type == DT\_REG) {

// Construct the full path of the file

char full\_path[256];

snprintf(full\_path, sizeof(full\_path), "%s/%s", source\_directory, entry->d\_name);

// Start a thread to encrypt this file

pthread\_create(&threads[thread\_count], NULL, encrypt\_thread, strdup(full\_path));

thread\_count++;

}

}

closedir(dir);

// Wait for all threads to finish

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

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

}

return 0;

}

```

In this modified code, we use `opendir` to open the source directory, and then `readdir` to iterate through its contents. For each regular file (not a directory), we start a new thread to encrypt the file. The number of threads is limited to 6, as you indicated.

Remember that when passing the `full\_path` to the `encrypt\_thread` function, it's essential to duplicate the string using `strdup`, as thread execution might continue while the main thread modifies the `full\_path` variable. Also, ensure you have proper error handling and memory management in your code, especially when dealing with duplicated strings and threads.

Compile the code with the `-pthread` flag to link the pthread library.