#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <pthread.h>

#include <unistd.h> // For fork and execve

#include <sys/wait.h> // For waitpid

#define MAX\_THREADS 6

#define BUF\_SIZE 1024

void\* encryptFile(void\* arg) {

// ... (same as before)

// Use fork and execve to run the encryption executable

pid\_t pid = fork();

if (pid == 0) { // Child process

char \*argv[] = {"./aes\_encrypt", inputFile, outputFile, NULL};

execve(argv[0], argv, NULL);

exit(1); // In case execve fails

} else if (pid > 0) { // Parent process

int status;

waitpid(pid, &status, 0);

if (WIFEXITED(status) && WEXITSTATUS(status) == 0) {

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

} else {

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

}

} else { // Fork error

perror("Fork error");

}

return NULL;

}

int main() {

// ... (same as before)

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

threadIndex++;

if (threadIndex >= MAX\_THREADS) {

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

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;

}

void\* encryptFile(void\* arg) {

// ... (same as before)

// Perform any necessary preparation before fork

// Open files, allocate memory, etc.

// Use fork and execve to run the encryption executable

pid\_t pid = fork();

if (pid == 0) { // Child process

char \*argv[] = {"./aes\_encrypt", inputFile, outputFile, NULL};

execve(argv[0], argv, NULL);

exit(1); // In case execve fails

} else if (pid > 0) { // Parent process

int status;

waitpid(pid, &status, 0);

if (WIFEXITED(status) && WEXITSTATUS(status) == 0) {

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

} else {

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

}

} else { // Fork error

perror("Fork error");

}

// Perform any necessary cleanup after fork

// Close files, free memory, etc.

return NULL;

}

int main() {

// ... (same as before)

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

threadIndex++;

if (threadIndex >= MAX\_THREADS) {

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

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;

}

int main() {

// Initialize variables and perform necessary preparations

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

// Check if input folder exists and is accessible

if (access(inputFolder, F\_OK) == -1) {

perror("Input folder does not exist");

return 1;

}

// Initialize OpenSSL (if needed)

// OpenSSL\_add\_all\_algorithms();

DIR \*dir;

struct dirent \*ent;

// ... (rest of your code)

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

// ... (rest of your code)

closedir(dir);

} else {

perror("Error opening directory");

return 1;

}

// Perform any necessary cleanup or finalization

// Cleanup OpenSSL resources (if needed)

// free dynamically allocated memory

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

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <pthread.h>

#include <unistd.h>

#include <sys/wait.h>

#define MAX\_THREADS 6

#define BUF\_SIZE 1024

void\* encryptFile(void\* arg) {

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);

// ... (rest of your code)

// Use fork and execve to run the encryption executable

pid\_t pid = fork();

if (pid == 0) { // Child process

char \*argv[] = {"./aes\_encrypt", inputFile, outputFile, NULL};

execve(argv[0], argv, NULL);

exit(1); // In case execve fails

} else if (pid > 0) { // Parent process

int status;

waitpid(pid, &status, 0);

if (WIFEXITED(status) && WEXITSTATUS(status) == 0) {

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

} else {

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

}

} else { // Fork error

perror("Fork error");

}

// ... (rest of your code)

return NULL;

}

int main() {

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

// Check if input folder exists and is accessible

if (access(inputFolder, F\_OK) == -1) {

perror("Input folder does not exist");

return 1;

}

// Initialize OpenSSL (if needed)

// OpenSSL\_add\_all\_algorithms();

DIR \*dir;

struct dirent \*ent;

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

threadIndex++;

if (threadIndex >= MAX\_THREADS) {

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

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;

}

// Perform any necessary cleanup or finalization

// Cleanup OpenSSL resources (if needed)

// free dynamically allocated memory

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

return 0;

}

To remove the original files after they have been successfully encrypted, you can add a step in the `encryptFile` function to remove the original file once the encryption process is complete. Here's how you can modify the code to achieve this:

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <pthread.h>

#include <unistd.h>

#include <sys/wait.h>

#define MAX\_THREADS 6

#define BUF\_SIZE 1024

void\* encryptFile(void\* arg) {

// ... (same as before)

// Use fork and execve to run the encryption executable

pid\_t pid = fork();

if (pid == 0) { // Child process

char \*argv[] = {"./aes\_encrypt", inputFile, outputFile, NULL};

execve(argv[0], argv, NULL);

exit(1); // In case execve fails

} else if (pid > 0) { // Parent process

int status;

waitpid(pid, &status, 0);

if (WIFEXITED(status) && WEXITSTATUS(status) == 0) {

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

// Remove the original file

if (remove(inputFile) != 0) {

perror("Error removing original file");

}

} else {

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

}

} else { // Fork error

perror("Fork error");

}

// ... (same as before)

return NULL;

}

// Rest of the code remains the same

```

In the modified `encryptFile` function, after the encryption process is successfully completed, the original input file is removed using the `remove` function. If the removal is unsuccessful, a corresponding error message is printed.

Keep in mind that removing files is a potentially irreversible action, so you should handle errors carefully and possibly make backups of the original files before performing such operations, especially in a production environment.