Project 5: Parallel Zip

Daniil Komov, Igor Zimarev

Link to source code Github repository: https://github.com/IkomovONE/Parallel_Zip

For this mini-project, we made two C programs that compress/decompress files in RLE encoding using parallel multi-threading.

Program pzip.c takes in one or multiple files (works best with .txt files), compresses the file/files using parallel threads and outputs the compressed file.

Program punzip.c takes in one or multiple compressed files (works best with compressed .txt files), decompresses the file/files using parallel threads and outputs the decompressed file.

Files pzip.c and punzip.c are the files containing the code for both programs. Both the programs can be compiled with gcc using this command: “gcc -o pzip pzip.c” or “gcc -o punzip punzip.c”

The program has been designed to work primarily with .txt files, and is limited in a way, but it makes sense to test it with other different file extensions (for example to compress .c file)

The repository already includes both of the compiled programs. There are several ways to run the programs:

-1 argument:

prompt> ./pzip inputFile > outputFile.z

This prompt will make program use 1 input file to compress it into “outputFile.z”. It’s important to note that compressed file has to be specified, and it has to be with .z extension.

-Many arguments:

prompt> ./pzip inputFile1 inputFile2 ... > outputFile.z

This prompt will make program take few input files, and compress them into one output file. It’s important to note that while there may be several input files, they are all being written into 1 compressed output file with .z extension.

-punzip program:

prompt> ./punzip compressed_inputFile > decompressed_outputFile.txt

or

prompt> ./punzip compressed_inputFile compressed_inputFile2 > decompressed_outputFile.txt

Similar execution is for punzip program: there may be several files to decompress, but they are all going into one unified decompressed file. The input files have to be accessible by the program.

Error handling

Before using the file to compress/decompress, make sure you have file permissions set-up as Read & Write for the system.

- If the file is corrupt of unavailable, error will pop up “error: cannot open file ‘<file>’”.

- In case of internal memory allocation issue, a error message will appear “malloc failed”.

The program has no file size limitations, however, very large file may affect system performance or compression may fail due to system memory constraints.

Considerations

The project codebase has been developed based on different considerations:

Parallelizing the compression.

The program simply divides the files into several (as many as the number of threads) segments, and each segment is being treated by assigned to it thread. Each segment gets only 1 thread, and all segments are made equal since the segment size is determined as file size if divided by the number of threads. This way system handles the file using several threads at once.

Determining how many threads to create

The program reads the number of system processors available, then accordingly creates as many threads as the number of these processors.

Efficiently performing each piece of work.

All segments are equally divided, and simple command in a loop is used to encode the contents of each segment. Memory mapping is used to access the files and parallelization is included. Thus, program can be treated as efficiency-oriented.

Accessing the input file efficiently

To access input files the program simply opens the file, then uses mmap() command to map the file into the memory address space.

Use cases examples

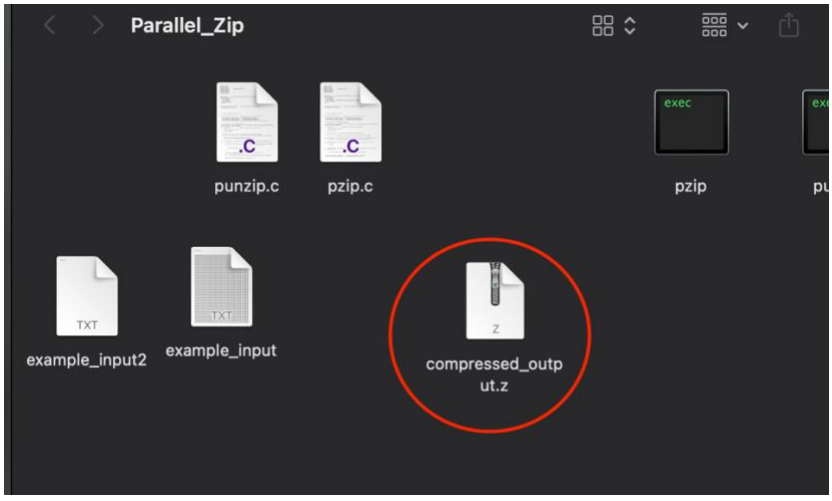
Repository contains example input files that are used here.

-1 argument:

prompt> ./pzip example_input.txt > compressed_output.z

```
@Daniils-MacBook-Air Parallel_Zip % ./pzip example_input.txt > compressed_output.z
@Daniils-MacBook-Air Parallel_Zip %
```

The program has no output in terminal, however, it creates the file inside the directory:



-Many arguments:

prompt> ./pzip example_input.txt example_input2.txt> compressed_output.z

The program never prints anything in terminal if the execution is successful. In this scenario the output of the program will be exactly same as in previous use case, except that the compressed_output.z file will have different contents.

-punzip:

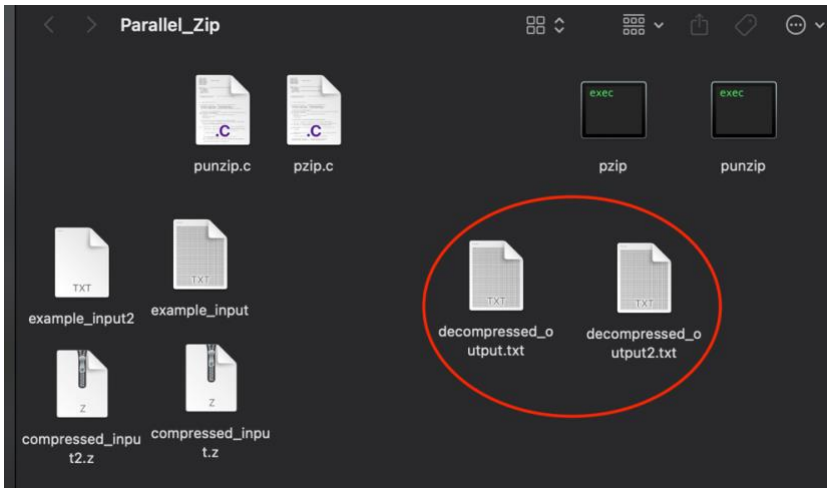
prompt> ./punzip compressed_input.z > decompressed_output.txt

or

prompt> ./punzip compressed_input.z compressed_input2.z > decompressed_output.txt

```
ikomovone@Daniils-MacBook-Air Parallel_Zip % ./punzip compressed_input.z > decompressed_output.txt
ikomovone@Daniils-MacBook-Air Parallel_Zip % ./punzip compressed_input.z compressed_input2.z> decompressed_output2.txt
ikomovone@Daniils-MacBook-Air Parallel_Zip %
```

The program executes the same way as pzip, except it will add decompressed file/files in the directory that would contain the original text:



Screenshots of the pzip.c code:

C pzip.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>          //importing libraries
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <pthread.h>



typedef struct {          //initializing certain data structure for each thread, with start of data segment, its length, and thread index
    char *seg_start;
    size_t seg_length;
    size_t th_index;
    char *output_buffer;   //Establishing output buffer and length for smooth result printing
    size_t output_length;
} Th_data;




void message_printer(const char *msg) {      //Function for printing errors and messages
    fprintf(stderr, "%s\n", msg);
}




//------Segment compression function--------

void *seg_compression(void *th_arg) {     //Establishing function for segment compression
```

```c

void *seg_compression(void *th_arg) {     //Establishing function for segment compression

    Th_data *seg_data = (Th_data *)th_arg;   //establishing thread data instance
    char *seg_start = seg_data->seg_start;        //initializing segment starting point
    size_t seg_length = seg_data->seg_length;  //initializing length of the segment


    //creating allocation for compressed segment data

    char *comp_output = malloc(seg_length * 2); // Allocating memory for the output, doubling the length of the segment so that there is always enoug

    if (!comp_output) {                //Adding error handling in case allocation fails

        message_printer("malloc failed");

        pthread_exit(NULL);
    }

    size_t comp_buff_position = 0; //initializing variable for positioning in the buffer


    for (size_t i = 0; i < seg_length;) {  //Using for-loop to go over each character inside the segment

        char character = seg_start[i]; //initializing the current character, setting it to the starting point in the segment
        size_t counter = 1;   //Initializing counter for RLE encoding. Counts the same character trend.


        while (i + counter < seg_length && seg_start[i + counter] == character) {  //Using while-loop to count characters of same type

            counter++;

        }

        comp_buff_position += sprintf(comp_output + comp_buff_position, "%zu%c", counter, character); //writing the data to the memory buffer
```

```
        comp_buff_position += sprintf(comp_output + comp_buff_position, "%zu%c", counter, character); //writing the data to the memory buffer


        i += counter; // Adjusting i variable for the loop to progress through the characters
    }


    seg_data->output_buffer = comp_output;        //Writing the output data to output buffer
    seg_data->output_length = comp_buff_position;   //Adjusting the position of buffer



    pthread_exit(NULL);  //exiting thread

}









//-------Main function-------



int main(int arg_counter, char *arg_select[]) {   //establishing the main function
```

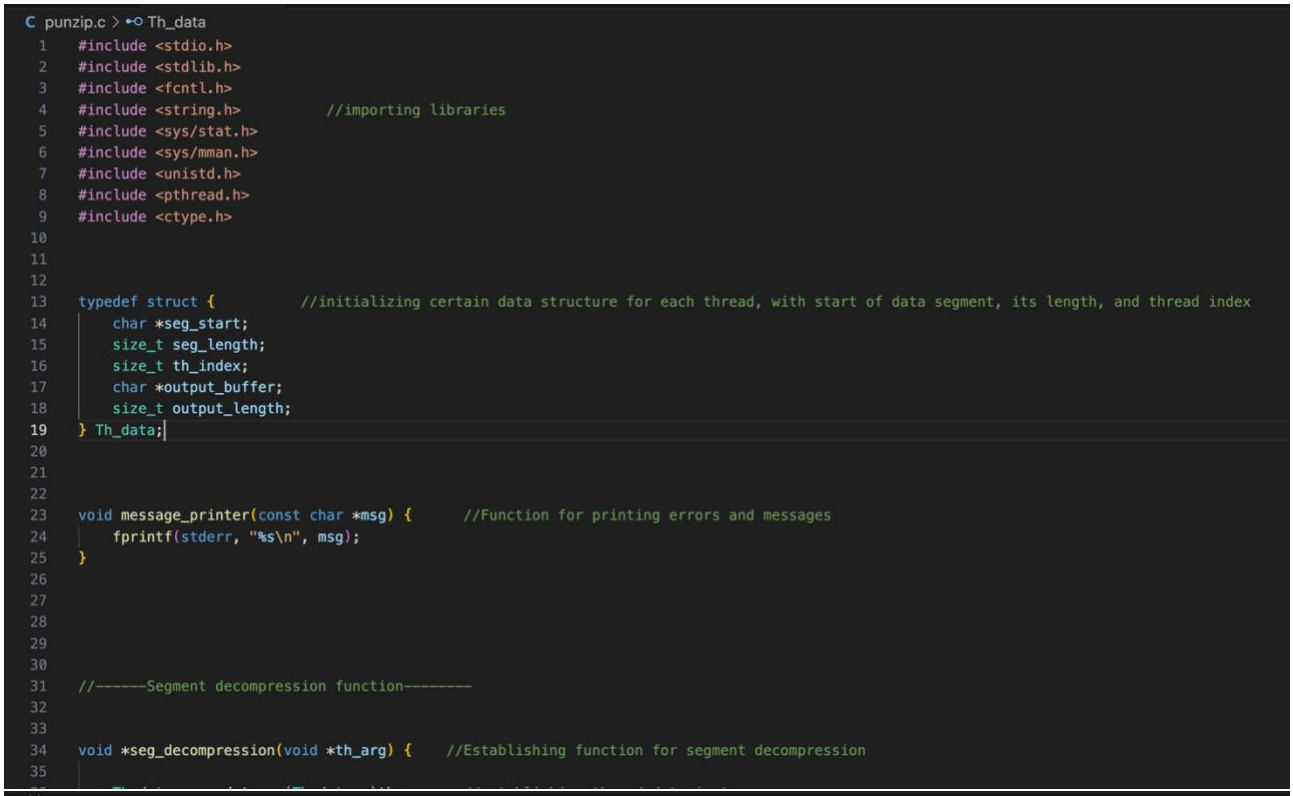
```


    if (arg_counter < 2) {           //Handling incorrect usage cases by throwing usage suggestion

        message_printer("usage: pzip <file1> <file2> ... > <outputFile>");  //calling message printer to print correct usage


        exit(1);   //exiting the program
    }



    //Next step is to establish the threads, doing it based on the number of processors system has

    int th_number  = sysconf(_SC_NPROCESSORS_ONLN);   //getting number of threads by getting the number of processors of the system. Using sysconf fu


    if (th_number < 1) {   //in case sysconf fails, we set the number of threads to 1 as default number (1 thread)
        th_number = 1;
    }

    pthread_t threads[th_number];   //Initializing the threads based on the established amount

    Th_data th_data[th_number];    //Establishing Th_data instances (amount of instances based on thread number). Th_data data structure has been in



    for (int i = 1; i < arg_counter; i++) {     //establishing a loop for all the arguments (input files)

        int input_file = open(arg_select[i], O_RDONLY);  //opening the file (one by one as it's a loop)
```

```c
for (int i = 1; i < arg_counter; i++) {     //establishing a loop for all the arguments (input files)

    int input_file = open(arg_select[i], O_RDONLY);  //opening the file (one by one as it's a loop)


    if (input_file < 0) {   //Error handling for file opening

        message_printer("Could not open the file");

        exit(1);
    }


    struct stat file_stats;   //initializing instance for collecting file statistics (for example size)



    if (fstat(input_file, &file_stats) != 0) {   //Error handling for file statistics

        message_printer("Couldn't get the size of the file");

        close(input_file);

        exit(1);
    }



    char *data = mmap(NULL, file_stats.st_size, PROT_READ, MAP_PRIVATE, input_file, 0);  //mapping file data into memory for easy access

    if (data == MAP_FAILED) {    //Error handling for file data mapping

        message_printer("Couldn't map the file data to memory");
```

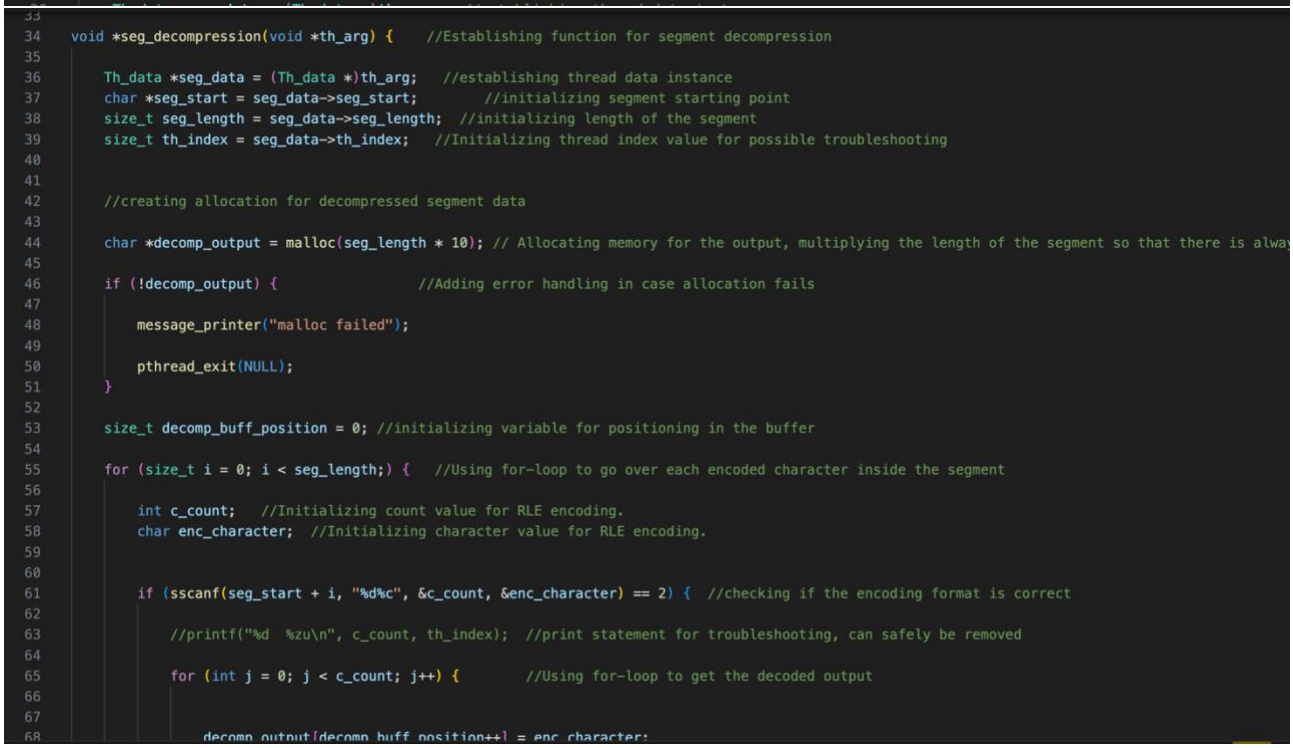
```c
            close(input_file);

            exit(1);
        }



        size_t seg_size = file_stats.st_size / th_number;  //establishing segments' sizes based on file size divided by number of threads, so that eac



        for (int k = 0; k < th_number; k++) {   //Using for-loop to create each thread 1 by 1, for file compression

            //Difining each element of the thread's data structure, such as start of segment, segment length and thread index


            th_data[k].seg_start = data + k * seg_size;   //Difining start of segment. The calculation is an offset that the thread has to make to get

            th_data[k].seg_length = (k == th_number - 1) ? (file_stats.st_size - k * seg_size) : seg_size; //Difining length of segment, includes last

            th_data[k].th_index = k;  //setting index of thread as k number of the for-loop (simple counting)


            pthread_create(&threads[k], NULL, seg_compression, &th_data[k]);  //creating thread, function for compressing the segment specified


        }
```

```
        }


        for (int j = 0; j < th_number; j++) {  //Using for-loop for joining all the threads
            pthread_join(threads[j], NULL);
            fwrite(th_data[j].output_buffer, sizeof(char), th_data[j].output_length, stdout); //Writing to the file

            free(th_data[j].output_buffer);   //Freeing the output buffer
        }


        munmap(data, file_stats.st_size);  //Freeing mapped memory
        close(input_file);  //closing input file
    }


    return 0;
}
```

Screenshots of the punzip.c code:



```c
C punzip.c > Th_data
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>          //importing libraries
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <pthread.h>
#include <ctype.h>



typedef struct {          //initializing certain data structure for each thread, with start of data segment, its length, and thread index
    char *seg_start;
    size_t seg_length;
    size_t th_index;
    char *output_buffer;
    size_t output_length;
} Th_data;



void message_printer(const char *msg) {      //Function for printing errors and messages
    fprintf(stderr, "%s\n", msg);
}




//------Segment decompression function--------


void *seg_decompression(void *th_arg) {    //Establishing function for segment decompression
```



```c
void *seg_decompression(void *th_arg) {    //Establishing function for segment decompression

    Th_data *seg_data = (Th_data *)th_arg;   //establishing thread data instance
    char *seg_start = seg_data->seg_start;        //initializing segment starting point
    size_t seg_length = seg_data->seg_length;  //initializing length of the segment
    size_t th_index = seg_data->th_index;   //Initializing thread index value for possible troubleshooting


    //creating allocation for decompressed segment data

    char *decomp_output = malloc(seg_length * 10); // Allocating memory for the output, multiplying the length of the segment so that there is alway

    if (!decomp_output) {                //Adding error handling in case allocation fails

        message_printer("malloc failed");

        pthread_exit(NULL);
    }

    size_t decomp_buff_position = 0; //initializing variable for positioning in the buffer

    for (size_t i = 0; i < seg_length;) {   //Using for-loop to go over each encoded character inside the segment

        int c_count;   //Initializing count value for RLE encoding.
        char enc_character;  //Initializing character value for RLE encoding.


        if (sscanf(seg_start + i, "%d%c", &c_count, &enc_character) == 2) {  //checking if the encoding format is correct

            //printf("%d  %zu\n", c_count, th_index);  //print statement for troubleshooting, can safely be removed

            for (int j = 0; j < c_count; j++) {        //Using for-loop to get the decoded output


                decomp_output[decomp_buff_position++] = enc_character;
```

```
            for (int j = 0; j < c_count; j++) {        //Using for-loop to get the decoded output


                decomp_output[decomp_buff_position++] = enc_character;


            }

            while (i < seg_length && isdigit(seg_start[i])) i++;  //Skip the count of the pair
            i++; // Skip the character from the pair

        } else {   //error handling for format errors. Throws error if program detects incorrect encoding format


            message_printer("Wrong encoding format");  //throwing the error

            pthread_exit(NULL);  //exiting the thread
        }
    }


    seg_data->output_buffer = decomp_output;     //Writing the result into the buffer
    seg_data->output_length = decomp_buff_position;   //Updating the buffer position



    pthread_exit(NULL);  //exiting thread
}



//-------Main function-------



int main(int arg_counter, char *arg_select[]) {   //establishing the main function


    if (arg_counter < 2) {           //Handling incorrect usage cases by throwing usage suggestion

        message_printer("usage: punzip <file1> <file2> ... > <outputFile1> <outputFile2>");  //calling message printer to print correct usage


        exit(1);   //exiting the program
    }



    //Next step is to establish the threads, doing it based on the number of processors system has

    int th_number  = sysconf(_SC_NPROCESSORS_ONLN);   //getting number of threads by getting the number of processors of the system. Using sysconf fun


    if (th_number < 1) {   //in case sysconf fails, we set the number of threads to 1 as default number (1 thread)
        th_number = 1;
    }


    pthread_t threads[th_number];   //Initializing the threads based on the established amount
```

```
134
135
136     pthread_t threads[th_number];   //Initializing the threads based on the established amount
137
138     Th_data th_data[th_number];    //Establishing Th_data instances (amount of instances based on thread number). Th_data data structure has been init
139
140
141
142
143
144     for (int i = 1; i < arg_counter; i++) {     //establishing a loop for all the arguments (input files)
145
146         int input_file = open(arg_select[i], O_RDONLY);  //opening the file (one by one as it's a loop)
147
148
149         if (input_file < 0) {   //Error handling for file opening
150
151             message_printer("Could not open the file");
152
153             exit(1);
154         }
155
156
157         struct stat file_stats;   //initializing instance for collecting file statistics (for example size)
158
159
160
161         if (fstat(input_file, &file_stats) != 0) {   //Error handling for file statistics
162
163             message_printer("Couldn't get the size of the file");
164
165             close(input_file);
166
167             exit(1);
171
172         char *data = mmap(NULL, file_stats.st_size, PROT_READ, MAP_PRIVATE, input_file, 0);  //mapping file data into memory for easy access
173
174         if (data == MAP_FAILED) {    //Error handling for file data mapping
175
176             message_printer("Couldn't map the file data to memory");
177
178             close(input_file);
179
180             exit(1);
181         }
182
183
184
185         size_t seg_size = file_stats.st_size / th_number;  //establishing segments' sizes based on file size divided by number of threads, so that e
186
187
188
189         for (int k = 0; k < th_number; k++) {
190
191             size_t seg_start_offset = k * seg_size;   //Getting start offset for the thread to know where to start
192             size_t seg_end = (k == th_number - 1) ? file_stats.st_size : seg_start_offset + seg_size;  //Determining the end of the segment
193
194
195
196             if (k > 0) {          //Using if-conditions and a loop to adjust start offset for the segments that are following after the 1st one
197
198                 while (seg_start_offset < seg_end && isdigit(data[seg_start_offset])) {
199                     seg_start_offset++;
200                 }
201
202                 if (seg_start_offset < seg_end) {    //Using if condition for the offset to move to a full pair or count-character
```

```
            if (seg_start_offset < seg_end) {    //Using if condition for the offset to move to a full pair or count-character
                seg_start_offset++;
            }
        }


        while (seg_end < file_stats.st_size && isdigit(data[seg_end])) {   //Using a loop and if condition to adjust the end of the segment as we
            seg_end++;
        }
        if (seg_end < file_stats.st_size) {
            seg_end++;
        }



        if (seg_start_offset >= file_stats.st_size || seg_end > file_stats.st_size) {  //Adding error handling for segment boundaries
            message_printer("Out of bounds error in segment. The compressed file may be corrupt");
            exit(1);
        }

        //Establishing thread data for creating the thread
        th_data[k].seg_start = data + seg_start_offset;
        th_data[k].seg_length = seg_end - seg_start_offset;
        th_data[k].th_index = k;

        //printf("Thread %d: seg_start_offset = %zu, seg_end = %zu, seg_length = %zu\n", k, seg_start_offset, seg_end, th_data[k].seg_length);

        pthread_create(&threads[k], NULL, seg_decompression, &th_data[k]);  //Creating thread
    }
```

```
    for (int j = 0; j < th_number; j++) {  //Using for-loop for joining all the threads


        pthread_join(threads[j], NULL);

        fwrite(th_data[j].output_buffer, sizeof(char), th_data[j].output_length, stdout);  //Writing to a file

        free(th_data[j].output_buffer);   //Freeing the output buffer

    }



    munmap(data, file_stats.st_size);  //Freeing mapped memory

    close(input_file);  //closing input file
}




return 0;

}
```