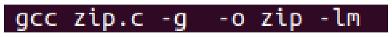
Operating Systems

Project Report

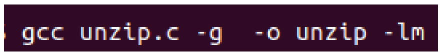
Code Usage:

Compile the code using the following command for zip file:



```
gcc zip.c -g  -o zip -lm
```

And for the unzip file:

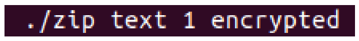


```
gcc unzip.c -g  -o unzip -lm
```

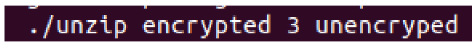
The code is executed with the following syntax:

```
zip <text file name> <number of threads> <output file name>
unzip <zipped file name> <number of threads> <output file name>
```

Example:



```
./zip text 1 encrypted
```



```
./unzip encrypted 3 unencryped
```

Methodology:

Zip File

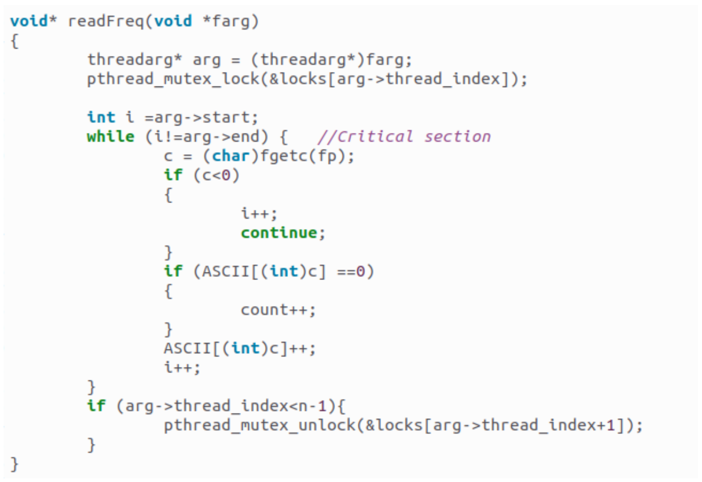
- The input file is first scanned for their frequencies of all unique values. The values are placed in an array where the index is the ASCII value of the character. This part is done using threads and each thread is assigned a section of the file to read.
- Then all the unique values are placed in a Node struct array and sorted in descending order based on their frequencies.
- A Huffman tree is formed in parallel using threads and each leaf node of the tree represents a unique character. The node holds their character value, frequency and code. The tree is traversed from the top and each leaf node is assigned the code based on the path traveled from the root node by the Node pointer. During this part, the code of each character is also written into a binary file along with their values. The file is named: “<output file name>_encoding.bin”.
- To encode the input file, each character is read. For each character, the Huffman tree is traversed, and whenever a value matching the read character is found, their code is written on a new file (the output file). This part in done in parallel and each thread is assigned a chunk of the input file again.
- At the end of the code, all the allocated memory is free and opened files are closed.

Unzip

- The input values are stored and validated. All the required memory is allocated in the heap.
- The binary file containing the code of each unique character is read from the binary file containing the encoded information. The value is placed in a struct Node 2D array, where the index is the value of the ASCII character of the value. The row index value is the ASCII value of the character, and rest of the columns contain the Huffman code of the unique character.
- Now the encoded file is analyzed. Whenever a '\n' is detected, it means that the code for one value of the code is finished, and the code is scanned from the array containing the codes of all characters. If the codes match, the character value is written in the output file. This whole part is done using threads. Each thread is thread is assigned a chunk of the encoded file.

Critical Sections:

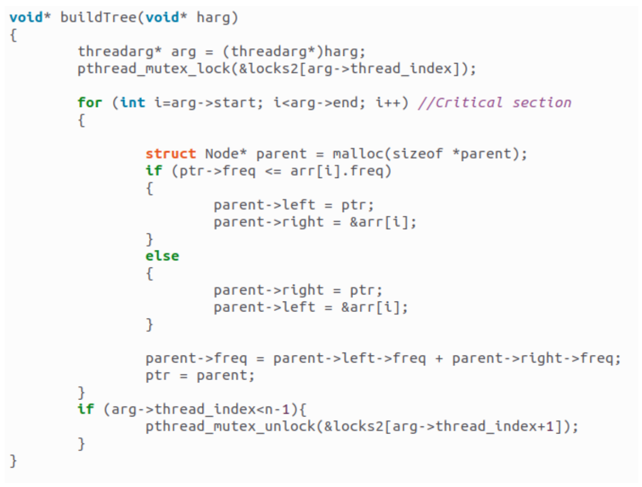
When the file is read to determine the frequency of each unique character:



```
void* readFreq(void *farg)
{
        threadarg* arg = (threadarg*)farg;
        pthread_mutex_lock(&locks[arg->thread_index]);

        int i =arg->start;
        while (i!=arg->end) {   //Critical section
                c = (char)fgetc(fp);
                if (c<0)
                {
                        i++;
                        continue;
                }
                if (ASCII[(int)c] ==0)
                {
                        count++;
                }
                ASCII[(int)c]++;
                i++;
        }
        if (arg->thread_index<n-1){
                pthread_mutex_unlock(&locks[arg->thread_index+1]);
        }
}
```

During the encoding of each character and building of the Huffman tree:



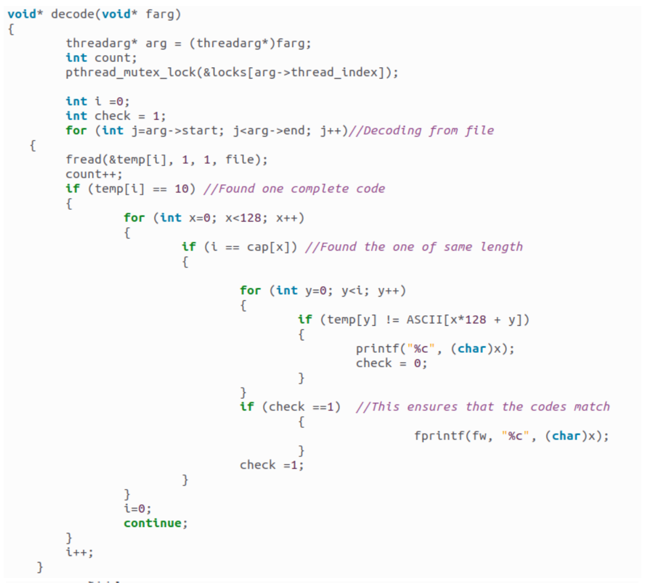
```
void* buildTree(void* harg)
{
        threadarg* arg = (threadarg*)harg;
        pthread_mutex_lock(&locks2[arg->thread_index]);

        for (int i=arg->start; i<arg->end; i++) //Critical section
        {

                struct Node* parent = malloc(sizeof *parent);
                if (ptr->freq <= arr[i].freq)
                {
                        parent->left = ptr;
                        parent->right = &arr[i];
                }
                else
                {
                        parent->right = ptr;
                        parent->left = &arr[i];
                }

                parent->freq = parent->left->freq + parent->right->freq;
                ptr = parent;
        }
        if (arg->thread_index<n-1){
                pthread_mutex_unlock(&locks2[arg->thread_index+1]);
        }
}
```

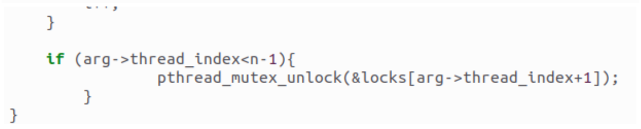
While decoding and unzipping the compressed file:



```
void* decode(void* farg)
{
        threadarg* arg = (threadarg*)farg;
        int count;
        pthread_mutex_lock(&locks[arg->thread_index]);

        int i =0;
        int check = 1;
        for (int j=arg->start; j<arg->end; j++)//Decoding from file
   {
        fread(&temp[i], 1, 1, file);
        count++;
        if (temp[i] == 10) //Found one complete code
        {
                for (int x=0; x<128; x++)
                {
                        if (i == cap[x]) //Found the one of same length
                        {

                                for (int y=0; y<i; y++)
                                {
                                        if (temp[y] != ASCII[x*128 + y])
                                        {
                                                printf("%c", (char)x);
                                                check = 0;
                                        }
                                }
                                if (check ==1)  //This ensures that the codes match
                                        {
                                                        fprintf(fw, "%c", (char)x);
                                        }
                                check =1;
                        }
                }
                i=0;
                continue;
        }
        i++;
    }
```

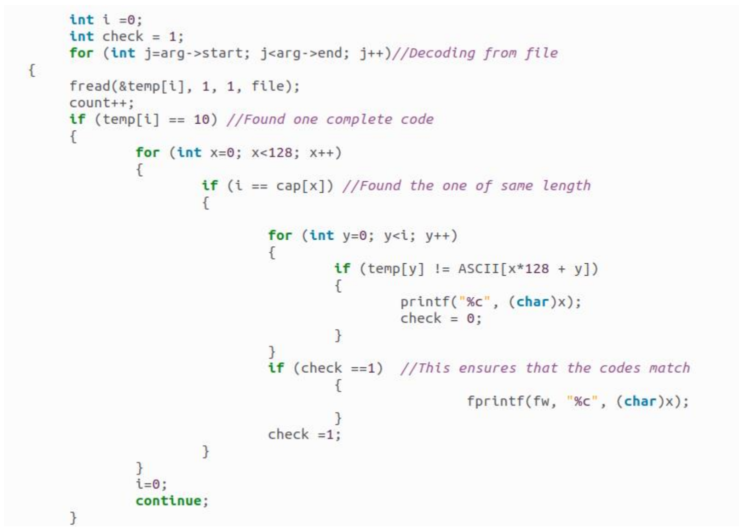


```
    }

    if (arg->thread_index<n-1){
                pthread_mutex_unlock(&locks[arg->thread_index+1]);
        }
}
```

Optimizations:

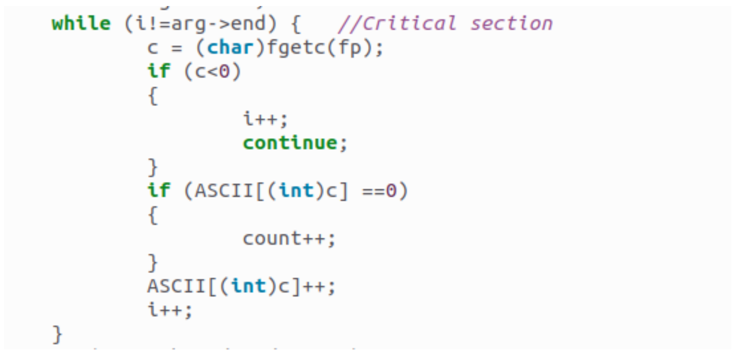
During the decoding stage, the code is scanned, and stored in an array. Another 2D array contains the codes of all the ASCII characters if it exists. Each row of the 2D array is compared with temporary array. Normally all columns of 128 rows of 2D array would have to be analyzed in order to determine the character value referred to by the code. In order to speed up the process, the lengths are compared of each row of the 2D array with the temporary array. If the lengths match, then it could be the same value which is furthered matched. Otherwise, unnecessary comparisons are avoided.



```
        int i =0;
        int check = 1;
        for (int j=arg->start; j<arg->end; j++)//Decoding from file
   {
        fread(&temp[i], 1, 1, file);
        count++;
        if (temp[i] == 10) //Found one complete code
        {
                for (int x=0; x<128; x++)
                {
                        if (i == cap[x]) //Found the one of same length
                        {

                                for (int y=0; y<i; y++)
                                {
                                        if (temp[y] != ASCII[x*128 + y])
                                        {
                                                printf("%c", (char)x);
                                                check = 0;
                                        }
                                }
                                if (check ==1)  //This ensures that the codes match
                                        {
                                                        fprintf(fw, "%c", (char)x);
                                        }
                                check =1;
                        }
                }
                i=0;
                continue;
        }
```

During the zipping phase, the frequencies are stored in a 128-sized array and the index represents the ASCII value of the character. Otherwise, a struct would have been needed to store both the frequency and the character value.



```
while (i!=arg->end) {   //Critical section
        c = (char)fgetc(fp);
        if (c<0)
        {
                i++;
                continue;
        }
        if (ASCII[(int)c] ==0)
        {
                count++;
        }
        ASCII[(int)c]++;
        i++;
}
```

Similar optimization is used for the 2D array described in the previous example.