Design Document - Huffman Coding

Lev Teytelman

February 21, 2022

1 Program Description

This assignment will use the Huffman encoding algorithm in order to compress files and reduce their sizes. It consists of two main programs:

- `encode` - Encodes a file using Huffman encoding
- `decode` - Decodes a file that has been encoded

2 Included Files

- encode.c - The implementation of the `encode` program with a `main` function, flag processing, and Huffman tree construction/encoding.
- decode.c - The implementation of the `decode` program with a `main` function, flag processing, Huffman tree construction, and file decoding.
- test.c - The implementation of the `test` program, created to test functionality of specific parts of the program and ensure everything is working as intended.
- defines.h - Macros for constants and magic numbers for clarity.
- header.h - The definition for the `Header` struct, for use in the encoded file.
- node.h - The function and struct declarations of the `Node` abstract data type.
- node.c - The implementation of the `Node` functions and struct.
- pq.h - The function and struct declarations for the `PriorityQueue` abstract data type.
- pq.c - The implementation of the `PriorityQueue` struct and related functions.
- code.h - The function and struct declarations for the `Code` abstract data type.
- code.c - The implementation of the `Code` struct and related functions.

- io.h - The declaration for functions related to file input and output, mostly used to ensure bulk reads and writes.
- io.c - The implementations of the aforementioned I/O functions.
- stack.h - The function and struct declarations for the `Stack` abstract data type.
- stack.c - The implementation of the `Stack` and related functions.
- huffman.h - The function declarations for creating and modifying Huffman trees.
- huffman.c - The implementations of the Huffman-tree-related functions.
- Makefile - The file used to easily compile and link files into executable binaries.
- README.md - An explanation of how to build and run the `encode` and `decode` programs.
- DESIGN.pdf - This document, explaining how the program will work.

3 Structure

The `encode` program will, in the simplest form, perform the following steps:

1. Read the input into a histogram of characters.
2. Build a Huffman tree based on the values in the histogram.
3. Create and write a header to the output file.
4. Write the tree to the output file.
5. For each character in the input, find the corresponding path down the tree and write it to file.

Our `decode` program will do the following:

1. Read and verify the header.
2. Read and reconstruct the Huffman tree that was written to file.
3. Read the encoded characters one bit at a time and find the matching characters on the tree, writing them to the output file.

4 Error Handling

Environments can and will have errors, especially with memory management and files! We will need to handle these appropriately to ensure our program runs smoothly. Firstly, we need to ensure that `malloc` and `calloc` do not have issues. We do this by checking the return value: if `NULL` is returned, the program was unable to allocate enough memory and we should exit.

A second important check we need to make is that the program is able to open files; if a file can't be opened, we may have many issues further on in our program. The way we open files with syscalls, as required for this assignment, is `open` and `creat` (to create files). These functions will return -1 if they fail, so we can easily compare whether or not their return values are $== -1$ and exit with an error message if so. Similarly, we will need to check if the input is seekable or not for our `encode` program, as we need to iterate the file twice and therefore go back to the start. If it is not seekable, we copy it into a temporary file and use that instead, ensuring we `unlink` it afterwards.

5 Pseudocode

In order to not read our file into memory all at once (as with large files this would not work or be extremely inefficient), we will read 4096 bytes at a time until we hit the end of the file. This way, the number of bytes being used is never more than the size of the array (4096) and our memory usage stays comparatively low:

```
num_of_read_bytes = None
while ((num_of_read_bytes = read(input, array, 4096)) > 0):
    for i in range(num_of_read_bytes):
        // do something with said bytes
```

Since our encoding program needs to read our input file to create a histogram and to encode the bytes, we will need to read through it twice; this requires it to be seekable. In order to check whether the file is seekable, we will use the syscall `lseek` without actually changing our position in the file (by passing 0 for the offset and the `SEEK_CUR` macro to specify the action). We will then check whether that fails (returns -1, and if so, we will copy the data 4096 bytes at a time into a temporary file that we can seek from, in a similar manner to the above pseudocode:

```
temp_created = false
if (lseek(input, 0, SEEK_CUR) == -1):
    temp_file = create_file("tmp")
    temp_created = true
    // read bytes similarly to the code above
    num_of_read_bytes = None
    while ((num_of_read_bytes = read(input, array, 4096)) > 0):
```

```
            write(temp_file, array, 4096)
        close(input) // close non-seekable input
        close(temp_file) // reopen temp file for reading
        input = open("tmp", READ)
```

After using the temp file, we will need to delete it from the filesystem. We will do this by checking for `temp_created` and using the syscall `unlink`:

```
    if (temp_created):
        unlink("tmp")
```

When reading and writing our header in the context of the encoded file, we will need to convert it into bytes when writing, and back into a header when reading. We will do this by passing a pointer to the header, which will point to the start of the sequence of bytes in memory representing the header. Then we will simply write that amount of bytes to the file:

```
    Header h = None
    write(output, &h, sizeof(Header))
```

We reference the header as the function requires a byte array, and passing the header directly would not work. We then tell it how much of the memory to read, to ensure we don't go outside of the boundaries and write random garbage to file.

Reading the header will work similarly to writing, but this time we will read the header size in bytes from file:

```
    Header h = None
    read(input, &h, sizeof(Header))
```

The header will also require the input file's permissions, the tree size, and the file size. Obtaining the file permissions and size will require a call to `fstat`:

```
def get_permissions(int file):
    stat stats = None
    fstat(file, &stats)
    return stats.st_mode

def get_size(int file):
    stat stats = None
    fstat(file, &stats)
    return stats.st_size
```

When reading in the input, we will use `fchmod` to set the output file's permissions:

```
    Header h = None
    read(input, &h, sizeof(Header))
    fchmod(output, h.permissions)
```