Assignment 6 Design Document

Alex Yeh

March 10, 2023

1 Description of Assignment

This assignment is to implement two programs called “encode” and “decode” which perform LZ78 compression and decompression, respectively. My encode program can compress any file, text or binary. My decode program can decompress any file, text or binary, that was compressed with encode. Both operate on both little and big endian systems. Both perform read and writes in efficient blocks of 4KB.

2 Files to be included in directory “asgn6”:

1. encode.c
    - This C file contains the main() function for the encode program.
2. decode.c
    - This C file contains the main() function for the decode program.
3. trie.c
    - This C file is the source file for the Trie ADT.
4. trie.h
    - This header file specifies the interface for the Trie ADT.
5. word.c
    - This C file is the source file for the Word ADT.
6. word.h
    - This header file specifies the interface for the Word ADT.
7. io.c

- This C file is the source file for the I/O module.

8. io.h
    - This header file specifies the interface for the I/O module.
9. endian.h
    - This header file specifies the interface for the endianness module.
10. code.h
    - This header file contains macros for reserved codes.
11. Makefile
    - This file directs the compilation process of encode.c and decode.c.
12. README.md
    - This file is in Markdown format and describes how to use my program and Makefile. It also lists and explains the different command-line options that my programs accept and explanations of complains by scan-build.
13. DESIGN.pdf
    - This file is a PDF version of this design document for assignment 6. It describes my design and design process for my programs with pseudocode.
14. WRITEUP.pdf
    - This file is a PDF version of my writeup for assignment 6. It includes everything I learned from this assignment.

3 Pseudocode

3.1 trie.c

This function is a constructor for a TrieNode
TrieNode *trie_node_create(uint16_t index)
    Allocate memory for a new node
    Check if the node is NULL and return NULL if it is
    Set node's code to index
    Iterate through each child node
        Set each child node pointer to NULL
    Return node

This function is a destructor for a TrieNode
void trie_node_delete(TrieNode *n)
    Free the memory allocated for the node

This function initializes a Trie
TrieNode *trie_create(void)
    Create a new root node with the code EMPTY_CODE
    Check if the root is NULL and return NULL if it is
    Return root

This function resets a trie to just the root TrieNode
void trie_reset(TrieNode *root)
    Iterate through each child of the root
        Delete each child
        Set each of the root's children to NULL

This function deletes a sub-trie starting from the trie rooted at node n
void trie_delete(TrieNode *n)
    Iterate through each child of the node
        If the child is not NULL
            Delete the child
            Set child to NULL
    Delete the node

This function returns a pointer to the child node representing the symbol sym
TrieNode *trie_step(TrieNode *n, uint8_t sym)
    Check if the child node is null and return NULL if it is
    Return the child representing the symbol sym

3.2 word.c

This function constructs for a word where sysms is the array of symbols a Word represents
Word *word_create(uint8_t *syms, uint32_t len)
    Allocate memory for a word
    Check if word is NULL and return NULL if it is
    Allocate memory for an array of symbols
    If array of symbols is NULL, free the memory allocated for "word" and return NULL
    Set the length of the word
    Iterate through each symbol in the array of symbols
        Copy the symbols from the array of symbols to the word's array of symbols
    Return word

This function constructs a new Word from the specified Word, w, appended with a symbol, sym.
Word *word_append_sym(Word *w, uint8_t sym)
    Define variable len and initialize to 0
    If word is not NULL, set len to length of the word
    Increment len by 1
    Allocate memory for a new word
    Check if the new word is NULL and free memory allocated for new word and return NULL if it is
    Allocate memory for an array of symbols with length of the old word plus one
    If the array of symbols is NULL, return NULL
    Set the length of the word to the length of a word with the appended symbol
    Iterate through each symbol in the array of symbols
        Copy the symbols from the array of symbols to the word's array of symbols
    Append the symbol to the end of the array of symbols
    Return the word

This function is a destructor for a Word, w
void word_delete(Word *w)
    Free the memory allocated for the array of symbols
    Free the memory allocated for the word

This function creates a new WordTable, which is an array of Words.
WordTable *wt_create(void)
    Allocate memory for a new word table using calloc
    If the word table is NULL, return NULL
    Set the word at index EMPTY_CODE to the empty word, a string of length of zero
    Return word table

This function resets a WordTable, wt, to contain just the empty Word.
void wt_reset(WordTable *wt)
    Iterate through each word in the word table
        Check if the word is not the empty word and the word is not NULL
            Free the memory allocated for each word in the word table
            Set each word in the word table to NULL

This function deletes a WordTable, wt.
void wt_delete(WordTable *wt)
    Iterate through each word in the word table
        Check if the word is not NULL
            Free the memory allocated for each word in the word table
    Free the memory allocated for the word table

3.3 io.c

This function is used to perform reads.
int read_bytes(int infile, uint8_t *buf, int to_read)
    Define variable “bytes_read” for the number of bytes read
    Define variable “bytes” for the number of bytes
    While the number of bytes read is less than the number of bytes to read
        Set bytes to the number of bytes read from the input file
        If bytes equals 0
            Break
        If bytes equals -1
            Print error message
            Return -1
        Add bytes to number of bytes read
    Add number of bytes read to total_syms
    Return the number of bytes read

This function is used to perform writes.
int write_bytes(int outfile, uint8_t *buf, int to_write)
    Define variable “bytes_written” for the number of bytes written
    Define variable “bytes” for the number of bytes
    While the number of bytes written is less than the number of bytes to write
        Set bytes to the number of bytes written to the output file
        If bytes equals 0
            Break
        If bytes equals -1
            Print error message
            Return -1
        Add bytes to number of bytes written
    Add number of bytes written to total_bits
    Return the number of bytes written

This function reads in sizeof(FileHeader) bytes from the input file.
void read_header(int infile, FileHeader *header)
    Set int bytes_read variable to reading bytes from infile
    If bytes_read is not equal to the size of the file header
        Print error message and exit program
    If big_endian
        Swap magic of header and set to header’s magic
        Swap protection of header and set to header’s protection
    If magic of header is not equal to MAGIC, print error message and exit

This function writes sizeof(FileHeader) bytes to the output file.

void write_header(int outfile, FileHeader *header)
    Set int bytes_written variable to writing bytes to outfile
    If bytes written is not equal to the size of the file header, print error message and exit
    If big_endian
        Swap magic of header
        Swap protection of header

Create a buffer called “buf” with size of BLOCK

This function reads a symbol from the input file.
bool read_sym(int infile, uint8_t *sym)
    Create variable “current” for index of block and initialize to 0
    Create variable “size” for size of block and initialize to 0
    If current is equal to size
        Read in bytes from infile into buffer and set to size
        Reset current to 0
    If index of block is less than the size of the block
        Set the symbol to the symbol at the index of the block
        Increment the index of the block
        Return true
    Return false

Create static buffer called “write_buffer and static ”index“

This function is a helper function for write_pair
void write_bit(int outfile, uint8_t bit)
    If the ith bit is 1, set the ith bit of the buffer to 1
    Else, set the ith bit of the buffer to 0
    Increment index
    If the index is equal to BLOCK multiplied by 8
        Call write_bytes function and reset index
    Iterate through each bit in the symbol (8 bits)
        If the ith bit is 1, set the ith bit of the buffer to 1
        Else, set the ith bit of the buffer to 0
    Increment index
    If the index is equal to BLOCK multiplied by 8
        Call write_bytes function and reset index to 0

This function writes a pair (code and symbol) to the output file
void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen)
    For each bit in the code
        Call write_bit function

For each bit in the symbol
Call write_bit function

This function writes out any remaining pairs of symbols and codes to the output file.
void flush_pairs(int outfile)
Set bytes to index/8
If index of block is not divisible by 8
Increment bytes by 1
Write bytes to outfile

This function is a helper function for read_pair
bool read_bit(int infile, uint8_t *bit)
Set static int variable ”read_index“ to 0
Create static buffer called ”read_buffer“ with size of BLOCK
Create static int variable ”length“ and initialize to 0
If read_index is 0 or read_index is equal to length multiplied by 8
Read in bytes from infile into buffer and set to int variable called ”num_read_bytes
Set read_index to 0
Set length to num_read_bytes
If length is 0
Return false
Set bit to the read_index of the buffer
Increment read_index by 1
Return true

This function “reads” a pair (code and symbol) from the input file.
bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen)
Set uint8_t variable track_bit to 0
Initialize code to 0 before reading
For each bit in the code
Call read_bit function using track_bit
If track_bit is 1, set the ith bit of the code to 1
Else, set the ith bit of the code to 0
For each bit in the symbol
Call read_bit function using track_bit
If track_bit is 1, set the ith bit of the symbol to 1
Else, set the ith bit of the symbol to 0
If code is equal to stop code
Return false
Return true

Create buffer called “write_word_buffer” with size of BLOCK

Create static int variable “word_index” and initialize to 0

This function “writes” a pair to the output file.
void write_word(int outfile, Word *w)
Iterate through each symbol in the word
Set word_index of buffer to symbol
Increment word_index by 1
If word_index is equal BLOCK
Write bytes to outfile
Reset word_index to 0

This function writes out any remaining symbols in the buffer to the outfile.
void flush_words(int outfile)
Write bytes to outfile

3.4 encode.c

A.1 Compression

```
COMPRESS(infile, outfile)
 1  root = TRIE_CREATE()
 2  curr_node = root
 3  prev_node = NULL
 4  curr_sym = 0
 5  prev_sym = 0
 6  next_code = START_CODE
 7  while READ_SYM(infile, &curr_sym) is TRUE
 8      next_node = TRIE_STEP(curr_node, curr_sym)
 9      if next_node is not NULL
10          prev_node = curr_node
11          curr_node = next_node
12      else
13          WRITE_PAIR(outfile, curr_node.code, curr_sym, BIT-LENGTH(next_code))
14          curr_node.children[curr_sym] = TRIE_NODE_CREATE(next_code)
15          curr_node = root
16          next_code = next_code + 1
17      if next_code is MAX_CODE
18          TRIE_RESET(root)
19          curr_node = root
20          next_code = START_CODE
21      prev_sym = curr_sym
22  if curr_node is not root
23      WRITE_PAIR(outfile, prev_node.code, prev_sym, BIT-LENGTH(next_code))
24      next_code = (next_code + 1) % MAX_CODE
25  WRITE_PAIR(outfile, STOP_CODE, 0, BIT-LENGTH(next_code))
26  FLUSH_PAIRS(outfile)
```

Pseudocode for compression from the assignment 6 pdf which I followed to write my encode function

Create static void function called “program_usage” to display program help and usage
Create static uint8_t function called “bit_length” to calculate the bit length of a code

int main(int argc, char **argv)

Set default option to 0
Set default input file to encrypt to stdin
Set default output file to encrypt to stdout
Set default verbose to false
While loop while opt is not equal to -1
    Switch statement for opt
        Case 'v':
            Set verbose to true
            Break
        Case 'i':
            Open input file name from user input
            If input file is equal to -1
                Print error message
                Return 1
            Break
        Case 'o':
            Open output file name from user input
            If output file is equal to -1
                Print error message
                Return 1
            Break
        case 'h':
            Call program_usage function
            Return 0
        default:
            Print error message
            Return 1
Create a new struct stat
Get the stats of the input file
Create a new file header
Set the magic number of the header to MAGIC
Set the protection of the header to the protection of the input file
Set permissions of the output file
Write the header to the output file
I then translated lines 1-26 of the image of the pseudocode above to C code
If verbose is true
    Print compressed file size
    Print decompressed file size
    Print space saving
Close infile
Close outfile

Delete trie to free memory
Return 0

3.5 decode.c

A.2 Decompression

```
DECOMPRESS(infile, outfile)
 1  table = WT_CREATE()
 2  curr_sym = 0
 3  curr_code = 0
 4  next_code = START_CODE
 5  while READ_PAIR(infile, &curr_code, &curr_sym, BIT-LENGTH(next_code)) is TRUE
 6        table[next_code] = WORD_APPEND_SYM(table[curr_code], curr_sym)
 7        WRITE_WORD(outfile, table[next_code])
 8        next_code = next_code + 1
 9        if next_code is MAX_CODE
10              WT_RESET(table)
11              next_code = START_CODE
12  FLUSH_WORDS(outfile)
```

Pseudocode for compression from the assignment 6 pdf which I followed to write my decode function

Create static void function called “program_usage” to display program help and usage
Create static uint8_t function called “bit_length” to calculate the bit length of a code

int main(int argc, char **argv)
Set default option to 0
Set default input file to encrypt to stdin
Set default output file to encrypt to stdout
Set default verbose to false
While loop while opt is not equal to -1
Switch statement for opt
Case ’v’:
Set verbose to true
Break
Case ’i’:
Open input file name from user input
If input file is equal to -1
Print error message
Return 1
Break
Case ’o’:
Open output file name from user input
If output file is equal to -1
Print error message
Return 1

        Break
    case 'h':
        Call program_usage function
        Return 0
    default:
        Print error message
        Return 1
Initialize header to 0
Read header from input file
Set output file permissions to input file permissions
I then translated lines 1-12 of the image of the pseudocode above to C code
If verbose is true
    Print compressed file size
    Print decompressed file size
    Print space saving
Delete table
Close infile
Close outfile
Return 0

3.6 Makefile

Set CC equal to “clang”
Set CFLAGS equal to “-Wall -Werror -Wextra -Wpedantic”
Set “all” to encode and decode Set “encode” dependencies on object files encode.o, trie.o, word.o, and io.o
Set “decode” dependencies on object files decode.o, trie.o, word.o, and io.o
Set default rule for creating .o from .c files
Set rule for make clean
Set rule for make format