Jaden Liu

Zliu259@ucsc.edu

12/5/2020

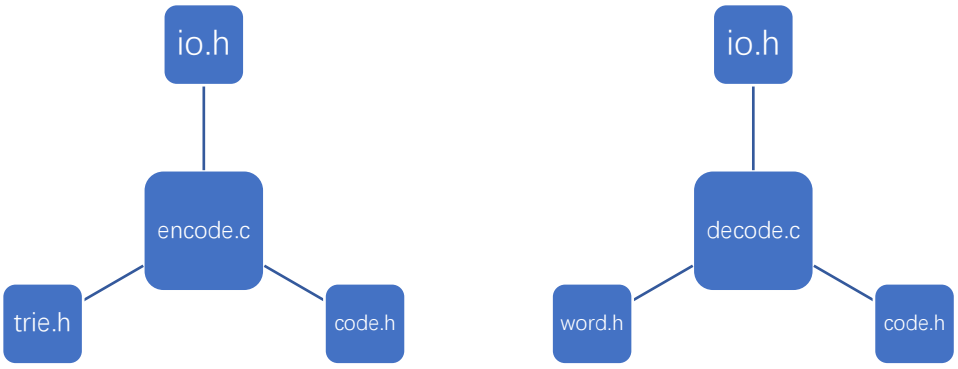
CSE13s Fall 2020

Assignment7:

Lempel-Ziv Compression

Design document

In this assignment, we need to have two main function: encode and decode, using the Lempel-Ziv compression method.



1. **Compression (cited from asgn7.pdf)**
    1. 1. Open infile with open(). If an error occurs, print a helpful message and exit with a status code indicating that an error occurred. infile should be stdin if an input file wasn't specified.
    2. The first thing in outfile must be the file header, as defined in the file io.h. The magic number in the header must be 0x8badbeef. The file size and the protection bit mask you will obtain using fstat(). See the man page on it for details.
    3. Open outfile using open(). The permissions for outfile should match the protection bits as set in your file header. Any errors with opening outfile should be handled like with infile. outfile should be stdout if an output file wasn't specified.
    4. Write the filled out file header to outfile using write_header(). This means writing out the struct itself to the file, as described in the comment block of the function.
    5. Create a trie. The trie initially has no children and consists solely of the root.

The code stored by this root trie node should be EMPTY_CODE to denote the empty word. You will need to make a copy of the root node and use the copy to step through the trie to check for existing prefixes. This root node copy will be referred to as curr_node. The reason a copy is needed is that you will eventually need to reset whatever trie node you've stepped to back to the top of the trie, so using a copy lets you use the root node as a base to return to.

6. You will need a monotonic counter to keep track of the next available code. This counter should start at START_CODE, as defined in the supplied code.h file. The counter should be a uint16_t since the codes used are unsigned 16-bit integers. This will be referred to as next_code.
7. You will also need two variables to keep track of the previous trie node and previously read symbol. We will refer to these as prev_node and prev_sym, respectively.
8. Use read_sym() in a loop to read in all the symbols from infile. Your loop should break when read_sym() returns false. For each symbol read in, call it curr_sym, perform the following:
    a) Set next_node to be trie_step(curr_node, curr_sym), stepping down from the current node to the currently read symbol.
    b) If next_node is not NULL, that means we have seen the current prefix. Set prev_node to be curr_node and then curr_node to be next_node.
    c) Else, since next_node is NULL, we know we have not encountered the current prefix. We buffer the pair (curr_node->code, curr_sym), where the bit-length of the buffered code is the bit-length of next_code. We now add the current prefix to the trie. Let curr_node->children[curr_sym] be a new trie node whose code is next_code. Reset curr_node to point at the root of the trie and increment the value of next_code.
    d) Check if next_code is equal to MAX_CODE. If it is, use trie_reset() to reset the trie to just having the root node. This reset is necessary since we have a finite number of codes.
    e) Update prev_sym to be curr_sym.
9. After processing all the characters in infile, check if curr_node points to the root trie node. If it does not, it means we were still matching a prefix. Buffer the pair (prev_node->code, prev_sym). The bit-length of the code buffered should be the bit-length of next_code. Make sure to increment next_code and that it stays within the limit of MAX_CODE. Hint: use the modulo operator.
10. Buffer the pair (STOP_CODE, 0) to signal the end of compressed output. Again, the bit-length of code buffered should be the bit-length of next_code.
11. Make sure to use flush_pairs() to flush any unwritten, buffered pairs.
12. Use close() to close infile and outfile.

2. Decompression (cited from asgn7.pdf)

1. Open infile with open(). If an error occurs, print a helpful message and exit with a status code indicating that an error occurred. infile should be stdin if an input file wasn't specified.
2. Read in the file header with read_header(), which also verifies the magic

number. If the magic number is verified then decompression is good to go and you now have a header which contains the original protection bit mask.

3. Open outfile using open(). The permissions for outfile should match the protection bits as set in your file header that you just read. Any errors with opening outfile should be handled like with infile. outfile should be stdout if an output file wasn't specified.
4. Create a new word table with wt_create() and make sure each of its entries are set to NULL. Initialize the table to have just the empty word, a word of length 0, at the index EMPTY_CODE. We will refer to this table as table.
5. You will need two uint16_t to keep track of the current code and next code. These will be referred to as curr_code and next_code, respectively. next_code should be initialized as START_CODE and functions exactly the same as the monotonic counter used during compression, which was also called next_code.
6. Use read_pair() in a loop to read all the pairs from infile. We will refer to the code and symbol from each read pair as curr_code and curr_sym, respectively. The bit-length of the code to read is the bit-length of next_code. The loop breaks when the code read is STOP_CODE. For each read pair, perform the following:
    a) As seen in the decompression example, we will need to append the read symbol with the word denoted by the read code and add the result to table at the index next_code. The word denoted by the read code is stored in table[curr_code]. We will append table[curr_code] and curr_sym using word_append_sym().
    b) Buffer the word that we just constructed and added to the table with buffer_word(). This word should have been stored in table[next_code].
    c) Increment next_code and check if it equals MAX_CODE. If it has, reset the table using wt_reset() and set next_code to be START_CODE. This mimics the resetting of the trie during compression.
7. Flush any buffered words using flush_words().
8. Close infile and outfile with close().

Pseudocode for compression and decompression

8.1 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          BUFFER_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      BUFFER_PAIR(outfile, prev_node.code, prev_sym, BIT-LENGTH(next_code))
24      next_code = (next_code + 1) % MAX_CODE
25  BUFFER_PAIR(outfile, STOP_CODE, 0, BIT-LENGTH(next_code))
26  FLUSH_PAIRS(outfile)
```

8.2 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      buffer_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)
```

Trie.h(cited from asgn7.pdf)

```c
#ifndef __TRIE_H__
#define __TRIE_H__

#include <inttypes.h>

#define ALPHABET 256

typedef struct TrieNode TrieNode;

//
// Struct definition of a TrieNode.
//
// children:  Each TrieNode has ALPHABET number of children.
// code:      Unique code for a TrieNode.
//
struct TrieNode {
  TrieNode *children[ALPHABET];
  uint16_t code;
};

//
// Constructor for a TrieNode.
//
// code:    Code of the constructed TrieNode.
// returns: Pointer to a TrieNode that has been allocated memory.
//
TrieNode *trie_node_create(uint16_t code);

//
// Destructor for a TrieNode.
//
// n:       TrieNode to free allocated memory for.
// returns: Void.
//
void trie_node_delete(TrieNode *n);

//
// Initializes a Trie: a root TrieNode with the code EMPTY_CODE.
//
// returns: Pointer to the root of a Trie.
//
TrieNode *trie_create(void);

//
// Resets a Trie to just the root TrieNode.
//
```

```
// root:    Root of the Trie to reset.
// returns: Void.
//
void trie_reset(TrieNode *root);

//
// Deletes a sub-Trie starting from the sub-Trie's root.
//
// n:       Root of the sub-Trie to delete.
// returns: Void.
//
void trie_delete(TrieNode *n);

//
// Returns a pointer to the child TrieNode reprsenting the symbol sym.
// If the symbol doesn't exist, NULL is returned.
//
// n:       TrieNode to step from.
// sym:     Symbol to check for.
// returns: Pointer to the TrieNode representing the symbol.
//
TrieNode *trie_step(TrieNode *n, uint8_t sym);

#endif
```

Trie_node_create(){
Malloc for for node
Set each children to NULL
Return node
}

Trie_node_delete(){
Free()
Set it to NULL
}

Trie_create(){
Root=trie_node_create();
Set the code to EMPTY_CODE
Return root
}

Trie_delete(){
For each children
    If children != NULL
        Trie_delete();
Free root
Root = NULL
}

```
Trie_reset()
For each children
        Trie_delete(children)
}

Trie_step(){
Return trie->children[sym]
}
```

Word.h(cited from asgn7.pdf)

```c
#include <inttypes.h>

//
// Struct definition of a Word.
//
// syms:  A Word holds an array of symbols, stored as bytes in an array.
// len:   Length of the array storing the symbols a Word represents.
//
typedef struct Word {
  uint8_t *syms;
  uint32_t len;
} Word;

//
// Define an array of Words as a WordTable.
//
typedef Word * WordTable;

//
// Constructor for a word.
//
// syms:    Array of symbols a Word represents.
// len:     Length of the array of symbols.
// returns: Pointer to a Word that has been allocated memory.
//
Word *word_create(uint8_t *syms, uint64_t len);

//
// Constructs a new Word from the specified Word appended with a symbol.
// The Word specified to append to may be empty.
// If the above is the case, the new Word should contain only the symbol.
//
// w:       Word to append to.
// sym:     Symbol to append.
// returns: New Word which represents the result of appending.
//
Word *word_append_sym(Word *w, uint8_t sym);



//
// Destructor for a Word.
//
// w:        Word to free memory for.
// returns: Void.
//
void word_delete(Word *w);

//
// Creates a new WordTable, which is an array of Words.
// A WordTable has a pre-defined size of MAX_CODE (UINT16_MAX - 1).
// This is because codes are 16-bit integers.
// A WordTable is initialized with a single Word at index EMPTY_CODE.
```

```
// This Word represents the empty word, a string of length of zero.
//
// returns: Initialized WordTable.
//
WordTable *wt_create(void);

//
// Resets a WordTable to having just the empty Word.
//
// wt:      WordTable to reset.
// returns: Void.
//
void wt_reset(WordTable *wt);

//
// Deletes an entire WordTable.
// All Words in the WordTable must be deleted as well.
//
// wt:      WordTable to free memory for.
// returns: Void.
//
void wt_delete(WordTable *wt);
```

Word_create(){
Malloc for word
Calloc for word->syms
Word->syms=syms
Word->len=len
Return word
}

Word_append_sym(){
Malloc for syms
Let syms=w->syms +sym
Word_create(new_word,syms)
Return new_word
}

Word_delete(){
Free()
Word=NULL
}

Wt_create(){
Malloc for wt
Set each word to NULL
Set the first one with empty string and 0 len
}

```
Wt_reset(){
Set each word =0;
}

Wt_delete(){
Free each word
Word=NULL
Free word table
Wt=NULL
}
```

code.h(cited from asgn7.pdf)

```
#ifndef __CODE_H__
#define __CODE_H__

#include <inttypes.h>

#define STOP_CODE   0               // Signals end of decoding/decoding.
#define EMPTY_CODE  1               // Code denoting the empty Word.
#define START_CODE  2               // Starting code of new Words.
#define MAX_CODE    UINT16_MAX      // Maximum code.

#endif
```

Io.h(cited from asgn7.pdf)

4.4 I/O

It is also a requirement that your programs, `encode` and `decode`, perform efficient I/O. Reads and writes will be done 4KB, or a block, at a time, which implicitly requires that you buffer I/O. Buffering is the act of storing data into a buffer, which you can think of as an array of bytes. Below is an I/O module that you are required to implement for the assignment.

```
#ifndef __IO_H__
#define __IO_H__

#include "word.h"
#include <inttypes.h>
#include <stdbool.h>

#define MAGIC 0x8badbeef     // Program's magic number.

//
// Struct definition of a FileHeader.
//
// magic:       Magic number indicating a file compressed by this program.
// protection:  Protection/permissions of the original, uncompressed file.
//
typedef struct FileHeader {
  uint32_t magic;
  uint16_t protection;
} FileHeader;

//
// Reads in sizeof(FileHeader) bytes from the input file.
// These bytes are read into the supplied FileHeader, header.
//
// infile:  File descriptor of input file to read header from.
// header:  Pointer to memory where the bytes of the read header should go
    .
// returns: Void.


//
void read_header(int infile, FileHeader *header);

//
// Writes sizeof(FileHeader) bytes to the output file.
// These bytes are from the supplied FileHeader, header.
//
// outfile: File descriptor of output file to write header to.
// header:  Pointer to the header to write out.
// returns: Void.
//
void write_header(int outfile, FileHeader *header);

//
// "Reads" a symbol from the input file.
// The "read" symbol is placed into the pointer to sym. (e.g. *sym = val)
// In reality, a block of symbols is read into a buffer.
// An index keeps track of the currently read symbol in the buffer.
// Once all symbols are processed, another block is read.
// If less than a block is read, the end of the buffer is updated.
// Returns true if there are symbols to be read, false otherwise.
//
// infile:  File descriptor of input file to read symbols from.
// sym:     Pointer to memory which stores the read symbol.
// returns: True if there are symbols to be read, false otherwise.
//
bool read_sym(int infile, uint8_t *sym);
```

```
55
56 //
57 // Buffers a pair. A pair is comprised of a code and a symbol.
58 // The bits of the code are buffered first, starting from the LSB.
59 // The bits of the symbol are buffered next, also starting from the LSB.
60 // The code buffered has a bit-length of bitlen.
61 // The buffer is written out whenever it is filled.
62 //
63 // outfile: File descriptor of the output file to write to.
64 // code     Code of the pair to buffer.
65 // sym:     Symbol of the pair to buffer.
66 // bitlen:  Number of bits of the code to buffer.
67 // returns: Void.
68 //
69 void buffer_pair(int outfile, uint16_t code, uint8_t sym, uint8_t bitlen);
70
71 //
72 // Writes out any remaining pairs of symbols and codes to the output file.
73 //
74 // outfile: File descriptor of the output file to write to.
75 // returns: Void.
76 //
77 void flush_pairs(int outfile);
78
79 //
80 // "Reads" a pair (code and symbol) from the input file.
81 // The "read" code is placed in the pointer to code (e.g. *code = val)
82 // The "read" symbol is placed in the pointer to sym (e.g. *sym = val).
83 // In reality, a block of pairs is read into a buffer.
84 // An index keeps track of the current bit in the buffer.
85 // Once all bits have been processed, another block is read.
86 // The first bitlen bits are the code, starting from the LSB.
87 // The last 8 bits of the pair are the symbol, starting from the LSB.
88 // Returns true if there are pairs left to read in the buffer, else false.
89 // There are pairs left to read if the read code is not STOP_CODE.
90 //
91 // infile:  File descriptor of the input file to read from.
92 // code:    Pointer to memory which stores the read code.
93 // sym:     Pointer to memory which stores the read symbol.
94 // bitlen:  Length in bits of the code to read.
95 // returns: True if there are pairs left to read, false otherwise.
96 //
97 bool read_pair(int infile, uint16_t *code, uint8_t *sym, uint8_t bitlen);
98
99 //
100 // Buffers a Word, or more specifically, the symbols of a Word.
101 // Each symbol of the Word is placed into a buffer.
102 // The buffer is written out when it is filled.
103 //
104 // outfile: File descriptor of the output file to write to.
105 // w:       Word to buffer.
106 // returns: Void.
107 //
108 void buffer_word(int outfile, Word *w);
109
110 //
111 // Writes out any remaining symbols in the buffer.
112 //
113 // outfile: File descriptor of the output file to write to.
114 // returns: Void.
115 //
116 void flush_words(int outfile);
117
118 #endif
```

```
Read_header(){
Read(infile,8byte)
}

Write_header(){
outfile,fileheader, 8byte
}

Read_sym(){
Read(infile, buf, 4096bytes)
For i,i++
*sym=buf[i]
Return *sym
}

Buffer pair(){
// Buffer code
For i=0;i<bit_len;i++{
If I <code_len
Copy the code to buffer
}
Padding with zeros
Write_index++
}

// buffer sym
For i=0;i<8;i++{
Copy sym
Write_index++
}

Read_pair(){
Read(infile, 4096)
// read code
For I<bitlen{
Copy code
Read_index++
}

// read sym
For I<8{
Copy sym
Read_index++
}
```

```
}
Buffer_word(){
For i<alphabet
Save to write_buf
If write_buf==4096bytes
Write_bytes(outfile,4096)
}

Flush_word(){
If (!STOP_CODE )
Write_bytes(write_index)
}
```