Assignment 6 WRITEUP.pdf

Roman Luo

March 12, 2023

1 tire.c

1.1 What I have learned:

- **Trie Structure:** A trie is a tree-like data structure used for efficient storage and retrieval of strings or other sequences of data. Each node in the trie represents a prefix of one or more strings. The trie is often used to implement string-related operations such as searching for a substring, finding the longest common prefix of a set of strings, and autocompletion. It is also used in data compression algorithms such as LZW compression, LZ78 for this assignment. Tries have several advantages over other data structures such as binary search trees and hash tables. They have a fast lookup time, typically O(n) where n is the length of the search string. They can also efficiently handle a large number of strings with a common prefix. However, they can have high space requirements, especially if the input alphabet is large.

- **TrieNode struct:** My code defines a trie data structure that uses a struct TrieNode to represent each node in the tree. The struct contains an array of TrieNode pointers, one for each possible symbol in the input ALPHABET, and a code value to assign to the node.

- **TrieNode functions:** The function trie_node_create() creates a new TrieNode with a given code value and returns a pointer to it. The function allocates memory for the node and its children using the malloc() function. It then sets the code value and initializes each child pointer to NULL. The function trie_node_delete() deletes a given TrieNode by checking if it is NULL then freeing the memory allocated for the node itself using the free() function.

- **Root TrieNode functions:** The functions trie_create() and trie_reset() are used to create and reset the root TrieNode. The function trie_create() creates the root TrieNode with a code value of EMPTY_CODE and returns a pointer to it. It calls trie_node_create() to create the node and checks if the allocation is successful before returning the pointer. The function trie_reset() deletes all children of the root TrieNode and frees the memory allocated for each child. It first checks if the root node pointer is NULL and returns if it is. It then calls trie_delete() on each child of the root before setting the child pointer to NULL.

- **Deleting TrieNodes:** The function trie_delete() deletes all TrieNodes in the trie recursively starting from a given node. It first checks if the node pointer is NULL and returns if it is. It then calls itself on each child of the node before calling trie_node_delete() to free the memory allocated for the node and its children.

- **TrieNode navigation:** The function trie_step() takes a pointer to the current node and a symbol as input and returns the pointer to the child node with the corresponding symbol. If the symbol is not present in the children, the function returns NULL.

2 word.c

2.1 What I have learned:

- **Word Structure:** The Word structure is a simple data structure used to represent a sequence of symbols. It consists of an array of uint8_t (syms) and a length field indicating the number of symbols in the array. Words are used extensively in text processing algorithms and data compression algorithms.

- **Word Functions:** The function word_create() creates a new Word with a given array of symbols and length. It allocates memory for the Word structure and the array of symbols using the malloc() function. It then copies the symbols from the input array to the new array and returns a pointer to the new Word. The function word_append_sym() creates a new Word by appending a symbol to an existing Word. It first creates a new array of symbols with the existing symbols and the new symbol, then creates a new Word with the new array and returns a pointer to the new Word. The function word_delete() frees the memory allocated for a Word structure and its array of symbols using the free() function.

- **WordTable Structure:** The structure consists of an array of Words and is indexed by codes. The first element of the array, at index EMPTY_CODE, is always the empty Word. Each subsequent element represents a sequence of symbols encoded by a code. WordTable structures are used to keep track of the codes and corresponding sequences of symbols generated during compression.

- **WordTable Functions:** The function wt_create() creates a new WordTable with a size of MAX_CODE. It allocates memory for the WordTable structure and initializes the first element, at index EMPTY_CODE, to the empty Word. The function wt_reset() resets a WordTable to contain only the empty Word. It deletes all other Words in the table and frees the memory allocated for them. The function wt_delete() deletes a WordTable and all its associated Words. It deletes each Word in the table and frees the memory allocated for them using the free() function.

3 io.c

3.1 What I have learned:

- **Buffer:** A buffer is a contiguous block of memory used to temporarily store data while it is being transferred from one location to another. A buffer can be thought of as a temporary holding area that allows data to be processed and manipulated before it is written to a file or transmitted over a network. In my case, I used 2 uint8_t buffers: uint8_t sym_buffer[BLOCK] for for read_sym, write_word, and flush_words functions, and uint8_t pair_buffer[BLOCK] for read_pair, write_pair, and flush_pairs functions as Ben Grant suggested on Piazza.

- **Static Index Tracking:** Unlike local variables, which are created and destroyed each time a function is called and returns, static variables are created and initialized only once, when the program starts, and remain in existence until the program terminates. Its space is allocated at compile-time rather than at run-time. I initialized 4 static int for tracking the buffer index since they retain their value between function calls.

```
static int read_pair_buffer_index = 0;
static int write_pair_buffer_index = 0;
static int sym_buffer_index = 0;
static int sym_buffer_index_end = 0;
```

- **Read/Write Bytes:** The function read(int fd, void *buf, size_t count) take a file descriptor fd, a buffer buf, and the number of bytes count. read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. Similarly, write(int fd, void *buf, size_t count) take a file descriptor fd, a buffer buf, and the number of bytes count. write() attempts to write up to count bytes from buffer buf into the file descriptor fd. Both read() and write() return the number of bytes they successfully proccessed.

- **Read/Write Headers:** Bother read_header and write_header functions call read_bytes and write_bytes to read or write the header from or to infile or outfile with the size of a user defined structure FileHeader. read_header also check if the bytes is in big endian and swap the magic number with swap32(), the protection number with swap16 from the functions that are defined in endian.h

```
static inline uint16_t swap16(uint16_t x) {
    uint16_t result = 0;
    result |= (x & 0x00FF) << 8;
    result |= (x & 0xFF00) >> 8;
    return result;
}

static inline uint32_t swap32(uint32_t x) {
    uint32_t result = 0;
    result |= (x & 0x000000FF) << 24;
    result |= (x & 0x0000FF00) << 8;
    result |= (x & 0x00FF0000) >> 8;
    result |= (x & 0xFF000000) >> 24;
    return result;
}
```

- **Read Pair:** read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen) read bitlen bits of a code into *code, and then a full 8-bit symbol into *sym, from infile. There are 2 parts, reading the code (iterating over bitlen), then the sym (iterating over 8). Both parts have identical structure, where it checks if the buffer is full by checking if index is BLOCK time 8; then proceed to write from the buffer to the outfile. The core of this function is to check if the current buffer is set with the AND operator, then set the corresponding code/sym to i-th bit. I got the if statement condition from the setbit function from bv16.h in the code comment repository, along with Ernani's help.

```
// return (x->v[k / BITS_PER_UNIT] >> k % BITS_PER_UNIT) & 0x1;
// extracts the value of the least significant bit of the shifted byte.
if (pair_buffer[read_pair_buffer_index / 8] >>
(read_pair_buffer_index % 8) & (uint8_t) 1) {
    // sets the ith bit of code to 1
    *code |= (1 << i);
}
```

- **Write Pair:** write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen) Write a pair – bitlen bits of code, followed by all 8 bits of sym – to outfile. There are 2 parts, writing the code (iterating over bitlen), then the sym (iterating over 8). Both parts have identical structure, where it checks if the buffer is full by checking if index is BLOCK time 8; then proceed to write from the buffer to the outfile. Tutor Varun also help me with setting buffer to zero after writing it using memset().

```
memset(pair_buffer, 0, BLOCK); //  Set buffer to all zeros. Varun
```

The core of this function is to check if code/sym is set with the AND operator, then calcuate the corresponding buffer index and set the specific bits. Here is an example, where write_pair_buffer_index over 8 calculates the buffer index and write_pair_buffer_index mod 8 calculates the amount to left shit to get the specific bit. The OR operator will insert the bit if its not 1.

```
// checks if the current bit of code is set
if (code & (1 << i)) {
```

```
        // set the corresponding bit in the buffer
        // write_pair_buffer_index / 8 calcuates buffer index
        // write_pair_buffer_index % 8 calcuates the amount to set for specific bit
        pair_buffer[write_pair_buffer_index / 8] |= (1 << write_pair_buffer_index % 8);
    }
```

- **Flush Pair:** flush_pairs(int outfile) writes any pairs that are in write_pair's buffer but haven't been written yet to outfile. Similar to write pairs, we use the index over 8 to calculate the buffer index, and if it can mod 8, we add 1 to the remaining bits.

```
    int remaining_bytes = write_pair_buffer_index
    / 8 + (write_pair_buffer_index % 8 ? 1 : 0);
```

4 encode, decode

4.1 What I have learned:

- **Open File:** Unlike fopen, open() opens a file and return a file descriptor but not a file stream. It takes in a file name and commands (you can use OR to pipe them) like the followings.
  - **O_RDONLY** - Open for reading only
  - **O_WRONLY** - Open for writing only
  - **O_RDWR** - Open for both reading and writing
  - **O_CREAT** - Create the file if it does not exist
  - **O_EXCL** - Fail if the file already exists
  - **O_TRUNC** - Truncate the file to zero length
  - **O_APPEND** - Append to the end of the file

  In my case, I only used O_RDONLY for reading the file. I used O_WRONLY — O_CREAT which will write to a file and create it if it does exit.
- **fstat:** The fstat() function in C is used to retrieve information about a given file descriptor. It takes a file descriptor as its argument and returns a struct stat containing various pieces of information about the file, such as its size, ownership, permissions, and modification time. The struct stat has the following fields:
  - st_dev: the ID of the device containing the file
  - st_ino: the file's inode number
  - st_mode: the file's type and permission bits
  - st_nlink: the number of hard links to the file
  - st_uid: the user ID of the file's owner
  - st_gid: the group ID of the file's owner
  - st_rdev: the device ID for special files
  - st_size: the size of the file in bytes
  - st_blksize: the optimal block size for I/O operations on the file
  - st_blocks: the number of 512-byte blocks allocated to the file
  - st_atime: the time the file was last accessed
  - st_mtime: the time the file was last modified
  - st_ctime: the time the file's inode was last changed

In my case, I only used st_mode for the protection bit as Ben Grant suggested.

```
FileHeader infile_header;
infile_header.magic = 0;
infile_header.protection = 0;

infile_header.magic = MAGIC;
struct stat protection_bits;
fstat(infile_descriptor, &protection_bits);
infile_header.protection = protection_bits.st_mode;
```