Sean Brandon
spbrando@ucsc.edu
11/29/2020

CSE13s Fall 2020
Assignment 7: Lempel-Ziv Compression
Design Document

Introduction:

In this assignment I am tasked with writing two programs: one named encode which takes a file and converts it into a compressed version of itself, and another named decode which takes in a file compressed by encode and it reverts it back into its original state. This will be done using the Lempel-Ziv compression algorithm which is lossless.

Pseudo-code:

io.c

Received help with functions in this file from pretty much every TA's section

```
// Buffers for reading and writing. Indexes for keeping track of where we are in the buffers
symbuf[4096]
symbuffindex = 0
bitbuf[4096]
bitbuffindex = 0

read_bytes (infile, buf, to_read) {
        Bytes_to_read = to_read
        Do {
                Bytes_read = read(infile, buf + total, bytes_to_read)
                Add bytes_read to total
                bytes _to_read = the number of bytes that haven't been read yet
        } while ((total isn't to_read) and (bytes read is greater than 0))
        total_syms = total    // This is for stats later
        Return total
}

write_bytes (outfile, buf, to_write) {
        Bytes_wrote = to_write
        Do {
                Bytes_write = write(outfile, buf + total, bytes_to_write)
                Add bytes_wrote to total
                bytes _to_write = the number of bytes that haven't been written yet
        } while ((total doesn't equal to_write) and (bytes_wrote is greater than 0))
        total_bits= total    // This is for stats later
        Return total
}
```

```
read_header (infile, header) {
        read_bytes(infile, header, sizeof(FileHeader))
        Bitbufindex += 8
        return
}

write_header (outfile, header) {
        write_bytes(outfile, header, sizeof(FileHeader))
        Bitbuffindex += 8
        return
}

read_sym (infile, byte) {
        if(symbuffindex == 0) {
                Num_of_bytes_read = read_bytes(infile, symbuf, 4096)
                if(num_of_bytes_read == 4096) {
                        Check = true
                }
                Else {
                        Temp =  num_of_bytes_read
                        Check = false
                }
        }
        byte = symbuf[symbuffindex]
        Symbuffindex = (symbuffindex + 1) % 4096
        return check OR (symbuffindex != temp)
}

buffer_pair (outfile, code, sym, bit_len) {
        Buffering sym
        for (i in 8) {
                If (sym & (1 << i % 8))) {
                        Set bit of bitbuffindex
                }
                Else {
                        Clear bit of bitbuffindex
                }
                Increment bitbuffindex
                If (bitbuffindex / 8 == 4096) {
                        write_bytes(outfile, bitbuf, 4096)
                        Bitbuffindex = 0
                }
        }
```

```
        // Buffering code
        for (i in bitlen) {
                If (code & (1 << i % 16))) {
                        Set bit of bitbuffindex
                }
                Else {
                        Clear bit of bitbuffindex
                }
                Increment bitbuffindex
                If (bitbuffindex / 8 == 4096) {
                        write_bytes(outfile, bitbuf, 4096)
                        Bitbuffindex = 0
                }
        }
        return
}

flush_pairs (outfile) {
        If (bitbuffindex is not 0) {
                If ((bitbufferindex & 8) is 0) {
                        Bytes = bitbuffindex / 8
                }
                Else {
                        Bytes = (bitbuffindex / 8) + 1
                }
                write_bytes(outfile, bitbuf, bytes)
        }
        return
}

read_pair (infile, code, sym, bit_len) {
        // Reading sym
        Code = 0
        for (i - 8) {
                if (bitbuffindex is 0) {
                        read_bytes(infile, bitbuf, 4096)
                }

                If (bitbuf at index / 8 anded with mask) {
                        Set bit of code at i
                }
                Else {
                        Clear bit of code at i
```

```
                }
                Bitbuffindex = (bitbuffindex + 1) % (4096 * 8)
        }

        // Reading code
        Code = 0
        for (i - bit_len) {
                if (bitbuffindex is 0) {
                        read_bytes(infile, bitbuf, 4096)
                }

                If (bitbuf at index / 8 anded with mask) {
                        Set bit of code at i
                }
                Else {
                        Clear bit of code at i
                }
                Bitbuffindex = (bitbuffindex + 1) % (4096 * 8)
        }

        Return (code != STOP_CODE)
}

buffer_word (outfile, w) {
        For (i - w->len) {
                Symbuf[symbuffindex++] = w->syms[i]
                If (symbuffindex = 4096) {
                        write_bytes(outfile, symbuf, 4096)
                        Symbuffindex = 0
                }
        }
        return
}

flush_words (outfile) {
        If (symbuffindex doesn't equal 0) {
                write_bytes(outfile, symbuf, symbuffindex)
                Symbuffindex = 0
        }
}
```

read_bytes is the wrapper for the read() syscall. This is how we read in bytes from a file to a buffer. Returns the number of bytes read.

write_bytes is similar to the previous function, read_bytes. It is the wrapper for the write() syscall. This is how we write out bytes from a buffer to a file. Returns the number of bytes written.
read_header reads in a FileHeader from the input file. Returns void
write_header writes a FileHeader to the output file. Returns void
read_sym reads a symbol from the input file. The read symbol is placed into the pointer to sym (passed by reference). 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.
buffer_pair buffers a pair. A pair consists of a symbol and a code. The bits of the symbol are buffered first, starting from LSB. The bits of the code are buffered next, also starting from the LSB. bit_len bits of the code are buffered to provide a minimal representation. The buffer is written out whenever it is filled. Returns void
flush_pairs writes out any remaining pairs of symbols and codes to the output file. Returns void.
read_pair reads a pair (symbol and code) from the input file. The read symbol is placed in the pointer to sym (pass by reference). The read code is placed in the pointer to code(pass by reference). In reality, a block of pairs is read into a buffer. An index keeps track of the current bit in the buffer. Once all bits have been processed, another block is read. The first 8 bits of the pair constitute the symbol, starting from the LSB. The next _bit_len bits constitutes the code, starting from the LSB. Returns true if there are pairs left to read in the buffer, else false. There are pairs left to read if the read code is not STOP_CODE.
buffer_word buffers a Word, or more specifically, the symbols of a Word. Each symbol of the Word is placed into a buffer. The buffer is written out when it is filled. Returns void.
flush_words writes out any remaining symbols in the buffer. Returns void.

trie.c

```
trie_node_create (index) {
        n = Malloc TrieNode
        If (n)
                For i in n->children
                        i = NULL
                n code = index
                Return n
        else
                printf(“error”)
                exit
}

trie_node_delete (n) {
        if (n)
                Free n
                n = NULL
```

```
        return
}

trie_create (void) {
        Return trie_node_create(EMPTY_CODE)
}

trie_reset (root) {
        For all n's in children
                If child exists
                        trie_delete(child)
        return
}

trie_delete (n) {
        If (n)
                For all child of n
                        trie_delete(child)
                trie_node_delete(n)
                N = NULL
}

trie_step(n, sym) {
        Return n->children[sym]
}
```

trie_node_create is the constructor for a TrieNode struct. Returns the constructed TrieNode
trie_node_delete is the destructor for a TrieNode struct. Returns null.
trie_create initializes a Trie: a root TrieNode with the index EMPTY_CODE (1). Returns the TrieNode root
trie_reset resets the entire Trie and all is children to NULL leaving only the root left. Returns void.
trie_delete recursively deletes a sub-Trie starting from th sub-Trie's root. Returns void.
trie_step Returns a pointer to the child TrieNode representing the input symbol sym. Returns NULL if the symbol doesn't exist.

word.c

```
word_create (syms, len) {
        w = malloc Word
        if (w)
                if (syms isnt NULL)
                        calloc for w syms
                        memcpy(w syms, syms, len)
                w len = len
```

```
                return w
}

word_append_sym (w, sym) {
        Make an array that is 1 larger than sym array
        For i in w len
                Array[i] = w->syms[i]
        Array last spot = sym
        Return word_create(array, w->len+1)
}

word_delete (w) {
        free(w->syms)
        free(w)
        return
}

wt_create (void) {
        wt = calloc WordTable
        wt[EMPTY_CODE] = word_create(NULL, 0)
        return wt
}

wt_reset (wt) {
        Loop through wt (array) calling word delete on everything but EMPTY_CODE
        return
}

wt_delete (wt) {
        Loop throughENTIRE wt (array) calling word delete on everything
        return
}
```

word_create is the constructor for a word struct. Returns a word.
word_append_sym constructs a new word from the given word appended with the given symbol and returns this new appended word.
word_delete is the destructor for a word. Returns void.
wt_create is the constructor for WordTable. WordTable is just an array of words of a predetermined length. Returns properly initialized WordTable.
wt_reset resets all indices in WordTable except for the empty word that it was initialized with. Returns void
wt_delete deletes an entire WordTable. Returns void.

encode.c

```
Includes
Extern variables
Prototype

Main {
        Set defaults for user inputs
        Stats = false
        Infile = STDIN_FILENO
        Outfile = STDOUT_FILENO
        Get_opt loop and switch

        Make the FileHeader
        FileHeader.magic = 0x8badbeef
        Struct stat header_protection
        fstat(infile, header_protection)
        FileHeader.protection = header_protection.st_mode
        fchmod(outfile, FileHeader.protection)
        write_header(outfile, FileHeader)

        encode pseudo-code given in lab pdf
        root = TRIE_CREATE()
        cur_node = root
        prev_node = NULL
        cur_sym = 0
        prev_sym = 0
        next_code = START_CODE
        while READ_SYM(infile, &curr_sym) is TRUE
                next_node = TRUE_STEP(curr_node, curr_sym)
                if next_node is not NULL
                        prev_node = curr_node
                        curr_node = next_node
                else
                        BUFFER_PAIR(outfile, curr_node.code, curr_sym,
        BIT-LENGTH(next_code))
                        curr_node.children[curr_sym] = TRIE_NODE_CREATE(next_code)
                        curr_node = root
                        next_code = next_code + 1
                if next_code is MAX_CODE
                        TRIE_RESET(root)
                        curr_node = root
                        next_code = START_CODE
                prev_sym = curr_sym
        if curr_node is not root
```

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

    Close files
    Return 0
}

// Helper function that finds the number of bits needed to represent a bit
bit_len(num) {
    while(num > 0) {
        count ++
        num / 2
    }
    Return count
}
```

decode.c

```
Includes
Extern variables
Prototype

Main {
    Set defaults for user inputs
    Stats = false
    Infile = STDIN_FILENO
    Outfile = STDOUT_FILENO
    Get_opt loop and switch

    Make FileHeader
    Read_header (infile, FileHeader)
    fchmod(outfile, FileHeader.protection
    If (FileHeader.magic != 0x8badbeef) {
        Return 0
    }

    decode pseudo-code given in lab pdf
    table = WT_CREATE()
    curr_sym = 0
    curr_code = 0
    next_code = START_CODE
    While READ_PAIR(infile, &curr_code, &curr_sym, BIT-LENGTH(next_code)) is TRUE
        table[next_code] = WORD_APPEND_SYM(table[curr_code], curr_sym)
```

```
                buffer_word(outfile, table[next_code])
                next_code = next_code + 1
                if next_code is MAX_CODE
                        WT_RESET(table)
                        next_code = START_CODE
        FLUSH_WORDS(outfile)

        Close files
        Return 0
}

// Helper function that finds the number of bits needed to represent a bit
bit_len(num) {
        while(num > 0) {
                count ++
                num / 2
        }
        Return count
}
```

Design Process:

I wrote trie.c and word.c first. Next I moved onto io.c which was a little tricky but with the help of the TA's I was able to complete it to the best of my abilities. However I don't think that io.c is totally right because my finished program unfortunately does not work properly. Anyways, after working on io.c I moved onto encode.c and decode.c which wasn't very difficult because almost the entirety of each were given in pseudo code. I only needed to add a getopt loop and some header stuff. After that I implemented stats which just required some externs. From there I went onto trying to debug everything I could but unfortunately I ran out of time and like I said my finished programs don't run correctly.