CSE13S Assignment 6 Writeup

Zion Kang

March 9 2023

1 Debugging / Testing

```
./test_trie
Running suite(s):
double free or corruption (top)
double free or corruption (top)
double free or corruption (top)
40%: Checks: 5, Failures: 0, Errors: 3
test_trie.c:11:P:trie_tests:test_trie_create:0: Passed
test_trie.c:34:E:trie_tests:test_trie_node_create:0: (after this point) Received signal 6 (Aborted)
test_trie.c:55:E:trie_tests:test_trie_node_delete:0: (after this point) Received signal 6 (Aborted)
test_trie.c:103:E:trie_tests:test_trie_step:0: (after this point) Received signal 6 (Aborted)
test_trie.c:79:P:trie_tests:test_trie_reset:0: Passed
make: [Makefile:21: all] Error 1 (ignored)
```

Figure 1: Trie Reset Bug

Upon running the test for trie provided by Dev, the errors above occurred. My initial thought process was to make print statements within the functions that didn't pass the tests which is what I did and found that my first implementation for trie_delete properly deleted the node n's children but not itself.

```c

// delete a sub-trie starting from trie rooted at node n
void trie_delete(TrieNode *n) {
    // since we're deleting a sub-trie starting from the trie rooted at n
    // we can simply use trie_reset() starting from the TrieNode n passed in
    //trie_reset(n);

    // recursively call on each of n's children and free them
    for (int i = 0; i < ALPHABET; i += 1) {
        if (n->children[i]) {
            // delete child node's children T_T
            trie_delete(n->children[i]);
            // delete child node
            trie_node_delete(n->children[i]);
            // set pointer to child node as NULL
            n->children[i] = NULL;
        }
    }
}
```

Figure 2: First Implementation of trie_delete

After moving the node delete outside of the loop checking for children, and an if statement at the outermost scope checking if the node n even exists, I was able to correctly delete n and it's children, seen in the 2nd implementation below.

```c

// delete a sub-trie starting from trie rooted at node n
void trie_delete(TrieNode *n) {
    // if n exists
    if (n) {
        // looping through n's children (0 - 255)
        for (int i = 0; i < ALPHABET; i += 1) {
            // if child exists at index i
            if (n->children[i]) {
                // recursive call to check if that child has more children
                trie_delete(n->children[i]);
            }
        }
        // once we hit node w/o children, delete node
        trie_node_delete(n);
        // set pointer to NULL
        n = NULL;
    }
}
```

Figure 3: Second Implementation of trie_delete

If our task at hand were only to delete n's children, the first implementation has the correct recursive calls to trie delete so as to check if child node has more children which would then check for more children then come back up to delete the node once it reached a node that didn't have more children. However, since we were calling it on n->children[i], we never deleted the node n itself. Which is why the second implementation works for our purpose of deleting n's children as well as itself.

Another small bug I ran into was the way I initialized the header, which resulted in a read error when trying to write the header, as can be seen below with the syscall write(buf) accessing uninitialized bytes.

```
zkang5@zkang5-VirtualBox:~/cse13snew/asgn6$ valgrind ./encode -i msg.txt -o out.txt
==4651== Memcheck, a memory error detector
==4651== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4651== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4651== Command: ./encode -i msg.txt -o out.txt
==4651==
got here in
got here out
got here header
==4651== Syscall param write(buf) points to uninitialised byte(s)
==4651==    at 0x4973D94: write (write.c:26)
==4651==    by 0x1098E2: write_bytes (in /home/zkang5/cse13snew/asgn6/encode)
==4651==    by 0x1099D8: write_header (in /home/zkang5/cse13snew/asgn6/encode)
==4651==    by 0x10A335: main (in /home/zkang5/cse13snew/asgn6/encode)
==4651==  Address 0x1ffefffe7e is on thread 1's stack
==4651==  in frame #3, created by main (???:)
==4651==
^C==4651==
```

Figure 4: Initialized Header Wrong

What seemed to fix this issue was simply initializing the header and setting its magic and protection bits in one line. (ex: FileHeader header = .magic = ..., .protection = ...). Based on this, I presume that when initializing the header, there were unitialized bytes, instead of them being 0s which led to this error. If I were to backtrack a little here, I ran into another issue while compiling my read and write header functions where I couldn't pass in my header buf to read the actual header data into because the read and write bytes functions accepted a param of (uint8_t *) not FileHeader *. To correct this issue, I first tried to type cast the header in a separate line above, but that threw an error. Then, I tried type casting the header within my call to read and write bytes, which compiled and function as desired.

```
zkang5@zkang5-VirtualBox:~/cse13snew/asgn6$ gcc -o io io.c
io.c: In function ‘read_header’:
io.c:62:12: warning: assignment to ‘FileHeader *’ from incompatible pointer type ‘uint8_t *’ {aka ‘unsigned char *’} [-Wincompatible
-pointer-types]
   62 |     header = (uint8_t *) header;
      |            ^
io.c:64:24: warning: passing argument 2 of ‘read_bytes’ from incompatible pointer type [-Wincompatible-pointer-types]
   64 |     read_bytes(infile, header, to_read);
      |                        ^~~~~~
      |                        |
      |                        FileHeader *
io.c:26:37: note: expected ‘uint8_t *’ {aka ‘unsigned char *’} but argument is of type ‘FileHeader *’
   26 | int read_bytes(int infile, uint8_t *buf, int to_read) {
      |                            ~~~~~~~~~^~~
io.c: In function ‘write_header’:
io.c:86:12: warning: assignment to ‘FileHeader *’ from incompatible pointer type ‘uint8_t *’ {aka ‘unsigned char *’} [-Wincompatible
-pointer-types]
   86 |     header = (uint8_t *) header;
      |            ^
io.c:88:26: warning: passing argument 2 of ‘write_bytes’ from incompatible pointer type [-Wincompatible-pointer-types]
   88 |     write_bytes(outfile, header, to_write);
      |                          ^~~~~~
      |                          |
      |                          FileHeader *
io.c:42:39: note: expected ‘uint8_t *’ {aka ‘unsigned char *’} but argument is of type ‘FileHeader *’
   42 | int write_bytes(int outfile, uint8_t *buf, int to_write) {
      |                              ~~~~~~~~~^~~
```

Figure 5: Passing in FileHeader * to (uint8_t *) param

When approach the io functions, the ones that I had the most trouble with were undoubtedly the pair functions which involved bitwise operators to get a certain bit in the code or sym. To test my functions, I used the test below which writes a single pair and prints the result out bit by bit from the file.

```
int main(void) {
    int outfile = open("out.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    uint16_t code = 27;
    uint8_t sym = 97;
    write_pair(outfile, code, sym, 7);
    for (uint32_t i = 0; i < 15; i += 1) {
        printf("%d", (pairs_buff[i/8] >> (i % 8)) & 1);
    }
    printf("\n");
    close(outfile);
    return 0;
}
```

Figure 6: Write Pair Test

```
zkang5@zkang5-VirtualBox:~/cse13snew/asgn6$ gcc -o io io.c
zkang5@zkang5-VirtualBox:~/cse13snew/asgn6$ ./io
code: got here, code_bit = 0
set bit in buff: 0
code: got here, code_bit = 1
set bit in buff: 1
code: got here, code_bit = 3
set bit in buff: 3
code: got here, code_bit = 4
set bit in buff: 4
sym: got here, sym_bit = 0
set bit in buff: 7
sym: got here, sym_bit = 5
set bit in buff: 12
sym: got here, sym_bit = 6
set bit in buff: 13
110110010000110
```

Figure 7: Write Pair Test Result

To get a better understanding of what's going on while the write pair function works, I also created print statements which check the code bit as well as the corresponding set bit in the buffer.

```c
// looping while there are bits in code left to write
while (1) {
    // check if code entire code has been buffered
    if (code_bit == bitlen) {
        break;
    }
    // if buffer is full
    if (pairs_index == BLOCK) {
        // flush the buffer
        flush_pairs(outfile);
    }
    // if bit in code is set
    if ((code & (1 << (code_bit % 16))) >> (code_bit % 16)) {
        // set bit in buffer
        printf("code: got here, code_bit = %u\n", code_bit);
        pairs_buff[bit_index/8] |= (1 << (bit_index % 8));
        printf("set bit in buff: %u\n", bit_index);
    }
    // inc bit count in code and bit index
    code_bit += 1;
    bit_index += 1;
}
```

Figure 8: Print Statements for Write Pair

After going through the same process for read pair, where I make print statements checking for the proper index within the code/sym as well as the buffer (seen above), I ran into other errors when trying to run my encode. For instance, I got an invalid read size of 4 error within my word_append_sym, which confused me very much because I had passed all the test for word and even tested the function myself pretty thoroughly.

As a result of this error, I decided to make a print statement for the results of the read pair function to check what word append sym was actually trying to access.

```c
// while there are pairs left to read
while (read_pair(infile, &curr_code, &curr_sym, get_bitlen(next_code))) {
    printf("curr code = %u, curr sym = %u, next code = %u\n", curr_code, curr_sym, next_code);
    // append read symbol to word noted by curr code and add result to table
    table[next_code] = word_append_sym(table[curr_code], curr_sym);
    // write word constructed above to outfile
    write_word(outfile, table[next_code]);
    // increment next code
    next_code += 1;
    // if we've reached max code, reset the wt
    if (next_code == MAX_CODE) {
        wt_reset(table);
        next_code = START_CODE;
    }
}
```

Figure 9: Print Statements for Read Pair

After checking that read pair was reading the wrong set of codes and syms, which were entirely different from what I had included in my test message file, I went through a similar debugging process for it as I did with my write pair, where I included print statements checking the bit indices within the code, sym, and buffer.

Even after fixing this issue, the hexdump results of my encoded file was different when compared to that of the binary's encoded file.

Note here that the message file was rather small and didn't exceed one block (4KB). Knowing this, I knew took a look at my flushpairs function, which was doing the actual writing to the outfile after reading in all the necessary codes and syms.

```c
// write out remaining pairs to the output file
void flush_pairs(int outfile) {
    // set bytes to flush
    int to_flush;
    if (bit_index % 8 == 0) {
        to_flush = bit_index / 8;
    }
    else {
        to_flush = (bit_index / 8) + 1;
    }
    // flush the toilet (from index 0 to curr index)
    write_bytes(outfile, pairs_buff, to_flush);
    // reset pairs buffer
    memset(pairs_buff, 0, BLOCK);
    // reset buff index
    bit_index = 0;
}
```

Figure 10: Flush Pair Bug

When calling on write_bytes, I was simply passing in the bit index within the buffer / 8, which in this case didn't work because C does floor division with the / operator. To work around this, I implemented a little bit of code above the write bytes call where I check if the bit index is a multiple of 8, and if so, divide the bit index by 8, otherwise divide it by 8 then add 1 to round up, and not down.

After getting the matching hexdumps as that of the binary, I tested with a larger test message and got the following:

```
zkang5@zkang5-VirtualBox:~/cse13snew/asgn6$ hexdump encoded.txt
0000000 baac baad 81b4 0000 55a1 3616 05be 6d24
0000010 6791 116e 6576 1696 2673 7216 2dca 8590
0000020 e0b7 ee66 5ba4 3338 9361 818c b490 36b0
0000030 9640 419b 6c1b c805 9791 2d84 4120 035a
0000040 8812 613d dbd8 1204 55dc 9767 045b 8242
0000050 6b68 db5c e6e0 1195 126c 0848 8532 698d
0000060 d8a5 16e9 a5bd 2b77 4408 b196 6b91 b2b0
0000070 da14 8d87 86d6 b100 71b5 70d2 bd72 ddb7
0000080 e4db 458c 4be6 94db da19 2062 b282 9bd9
0000090 4f33 66d4 739c 21c8 28dd 4a6f 1e90 8b88
00000a0 57ed 2ce6 960b 4145 6e6a 368a 882a ce49
00000b0 a20a c37a 6186 b440 4d69 0e90 9408 d40d
00000c0 7446 9703 e631 61aa b728 9bab 2405 669b
00000d0 db60 a482 84d1 526f 6290 1b99 1c44 0706
00000e0 bb21 c3c9 69cc 6974 6d68 684f 2077 615f
00000f0 7337 6e6f 6e3a 6771 204b 6b67 6e40 6c32
0000100 796b 6553 627a 6439 2015 677d 6989 6b80
0000110 208b 6c6a 7943 208e 7324 6486 7791 613c
0000120 207f 652f 6e5c 6c8c 2097 2079 2099 6e7b
0000130 2087 2092 2073 209e 6576 6d5e 616c 2085
0000140 64a4 6f9b 6e65 692d 6594 209f 6f96 6e78
0000150 206d 62a3 6f90 6e49 6e7e 20b0 6da8 6582
0000160 79aa 6998 6147 777c 6e9c 6cba 6d75 6cbc
0000170 208d 69a2 6ebf 77a5 64c1 6b4c 6895 6d83
0000180 6dab 6261 6eb7 2088 6ca7 65c3 6c5d 6ec5
0000190 69b5 64c7 6eae 64b9 65ca 68b2 79cc 73be
00001a0 6f9a 67b8 2093 65bb 6cd3 6ea1 73d5 6edd
00001b0 61d0 6c9d 6ed2 6fa0 61b4 62ac 20cf 61a6
00001c0 69e6 6581 6fe1 6de9 72ce 7763 20c9 658a
00001d0 65da 20bd 73c6 20e4 6eec 69c2 6eee 20e8
00001e0 6d84 61f1 6fc0 c8af 81b2 62a4 16a3 4ea6
00001f0 de4e 401b 7a10 0220 c4d9 1185 fb0c 86e7
0000200 e40d 1b60 b631 6d84 dcfe bd90 4c19 058b
0000210 f082 5e0d 23d9 82b9 0c6e 26d3 39b6 ab7c
0000220 6208 ee4b d9e2 b637 2073 40b3 95fa 0c59
0000230 785b 31e6 1f6d c808 66b6 2265 22dd 61ba
0000240 0104 c6e7 edaf 0824 b0c7 2091 d2d1 a24a
```

Figure 11: Hexdump of Encoded File from my implementation

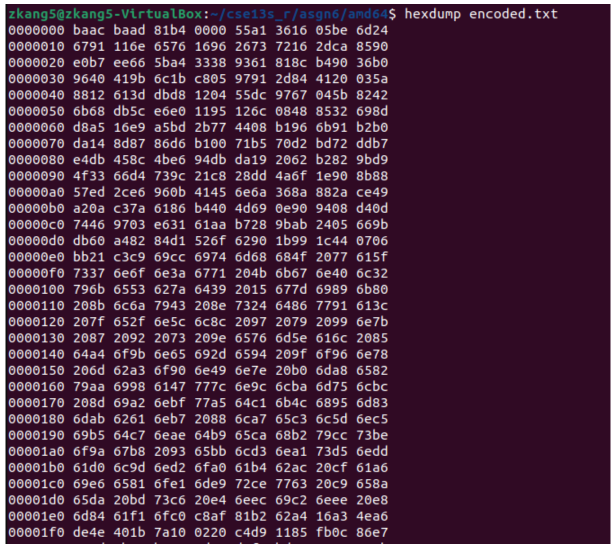

Figure 12: Hexdump of Encoded File from Binary

2 Findings

With testing different sizes of files, meaning different amounts of symbols within my test text files, I could see that with smaller files, compression was rather counter-intuitive because the compressed files were larger in size and led to more space being used up. On the other hand, when testing with a text file with 5000+ symbols, or characters, I had a high space saving percentage, where the compressed file size was much smaller than the uncompressed size.

Most of the above findings were clearly stated in the assignment document and were to be expected. However, I think something I found interesting was that it was a requirement in the assignment for the programs to be interoperable, meaning they can work with big and little endian systems. Since my virtual machine (Ubuntu) is little endian, I think it would've been interesting to be able to fully test this interoperability.