Daryl Stronge ID: 917844673
Github: DSnoNintendo CSC415.01 Operating Systems

Assignment 2 – Struct and Buffer

Description:
This assignment is to write a C program that instantiates a given struct using malloc and write it to memory. Write an algorithm to copy strings of unknown sizes to fix-sized buffers, and write the buffers to memory.

Approach:
The first step I took in this assignment was learning about memory allocation. This was essential in completing the assignment. The main memory methods I used in this assignment were *malloc*, *memcpy*, and *memset,* and *free*. *Malloc* was used to allocate memory, *memcpy* was used to copy strings to buffers, *memset* was used to clear allocated memory to be reused, and *free* was used to unallocate memory after it was used.

After learning about memory I was able to instantiate the *personalInfo* struct. After doing so, I was able to use malloc to allocate memory for the struct as well as the memory necessary for the firstName and lastName member variables.

After the struct was written to memory and the allocated memory was *free'*d, I needed to set up the buffer-writing algorithm. Memory in bytes equivalent to the defined *BLOCK_SIZE* needed to be allocated for the buffer and an integer variable needed to be instantiated (*buffer_pos*) with value of 0 to keep track of how many bytes were present in the buffer. Everytime a string is copied to the buffer, the length of the string added to the buffer will be added to *bufferPos* integer variable. Whenever memcpy is called, the first argument, *__dest*, would be set to:

```
buffer + buffer_pos
```

This returns a pointer to the next available byte in the buffer.

My next step was figuring out how to write segments of strings to the buffer if a string was too large to fit. I was able to figure this out through reading documentation on *memcpy*. Because there is an argument of *memcpy* that allows users to define the number of bytes to be copied (*__n*), if *__n* is set to the amount of free bytes in the buffer, only a substring of the string would be copied to the buffer. By setting the *__n* argument of *memcpy* to:
`BLOCK_SIZE - buffer_pos`,
a substring of the necessary size would be copied to the buffer whenever a string was too large. This is because the above code returns an integer equivalent to the amount of bytes remaining in the buffer.

My next step was learning how to copy the remaining bytes of the string to a new buffer. After the buffer has been filled to capacity, it is written to memory, and the buffer is reset using *memset*. The remaining characters that weren't added to the recently committed buffer are

written to the new buffer by setting the second argument of *memcpy, __src*, to:
str + (BLOCK_SIZE - buffer_pos)
This returns a substring that starts off where the last substring ended. The *__n* argument is set to:
string_length - (BLOCK_SIZE - buffer_pos)
which ensures that only the remaining characters are written to the buffer.

Issues and Resolutions:

My first issue was finding an efficient way to segment strings that were too large for the buffer. Initially I wrote a function that creates substrings of a desired length, but since this would require allocating extra memory, I tried to find another way to do this. Through reading more about the *memcpy* function, I was able to more efficiently copy substrings to the buffer using the dynamic programming approach explained above.

My next issue was figuring out when to use *free* when copying strings to buffers. At first, I used *free* in the middle of the loop and reallocated memory, but when new strings were added to the buffer, unexpected characters would be included. I also tried simply calling *malloc* whenever a block was committed. This worked, but upon doing more research I learned that this makes the previously allocated memory inaccessible, creating memory leaks. I learned that *memset* was the most efficient way to clear allocated memory for reuse.

Analysis:

```
END-OF-ASSIGNMENT
000000: 40 C7 C0 CE 4C B4 00 00  60 C7 C0 CE 4C B4 00 00 | @???L?..`???L?..
000010: C1 32 B5 36 14 00 00 00  12 00 04 00 46 6F 75 72 | ?2?6........Four
000020: 20 73 63 6F 72 65 20 61  6E 64 20 73 65 76 65 6E |  score and seven
000030: 20 79 65 61 72 73 20 61  67 6F 20 6F 75 72 20 66 |  years ago our f
000040: 61 74 68 65 72 73 20 62  72 6F 75 67 68 74 20 66 | athers brought f
000050: 6F 72 74 68 20 6F 6E 20  74 68 69 73 20 63 6F 6E | orth on this con
000060: 74 69 6E 65 6E 74 2C 20  61 20 6E 65 77 20 6E 61 | tinent, a new na
000070: 74 69 6F 6E 2C 20 63 6F  6E 63 65 69 76 65 64 20 | tion, conceived
```

The above hexdump represents the personalInfo struct that was written to memory.

The 8 bytes highlighted yellow represent the memory address of the *firstName* string. The string lives at the memory address 0xB44CEC0CC740. I was able to verify this by printing the memory address of *firstName*. This address will be different every time the program is run.

```
 printf("firstName pointer: %p\n", pi_ptr->firstName);
```

The remaining 00 bytes represent unoccupied space in the char pointer, which is typically 8 bytes.

The bytes highlighted green represent the memory address to the *lastName* string. I was able to verify this using the same code shown above.

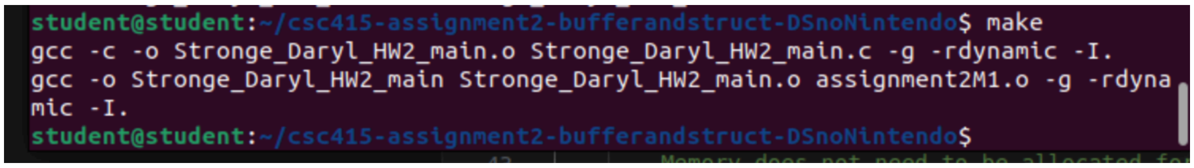
The bytes highlighted pink represent the *studentID* integer. *StudentID* (917844673) converted to hexadecimal is **36B532C1**. This is represented in the 4 highlighted bytes, but stored backwards in this hexdump format.

The 4 bytes highlighted purple represent the *gradeLevel* enum value as an integer. The enum value *SENIOR* is mapped to the decimal value 20, which is **14** when converted to an integer. The remaining 00 bytes represent unoccupied space in the integer, which is 4 bytes. All bytes were occupied for studentID, because the number was big enough to do so.

The 4 bytes highlighted blue represent the *languages* integer field of the personalInfo struct as a hexadecimal value. 262162 converted from decimal to hexadecimal is 00040012, which is represented backwards in the highlighted portion of the hexdump.

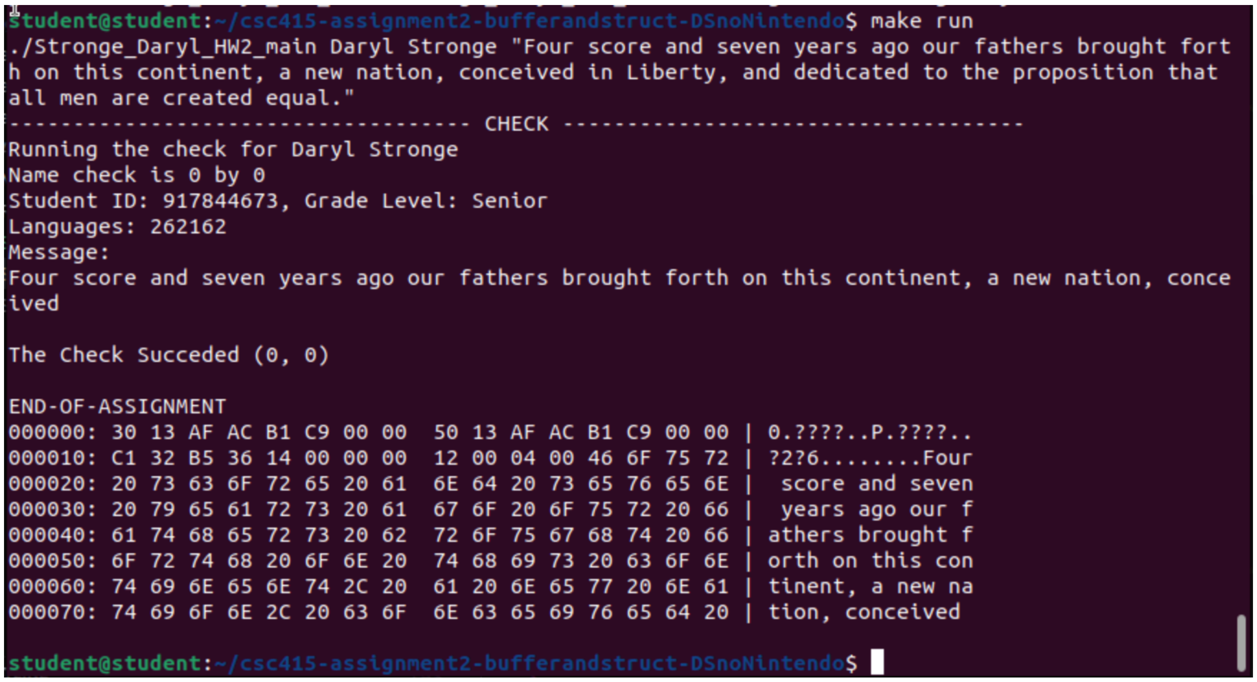
The remaining unhighlighted bytes represent the *message* character array. Because the string was stored using a character array, the bytes are displayed as ASCII values in this portion of the hexdump.

Screen shot of compilation:



```
student@student:~/csc415-assignment2-bufferandstruct-DSnoNintendo$ make
gcc -c -o Stronge_Daryl_HW2_main.o Stronge_Daryl_HW2_main.c -g -rdynamic -I.
gcc -o Stronge_Daryl_HW2_main Stronge_Daryl_HW2_main.o assignment2M1.o -g -rdyna
mic -I.
student@student:~/csc415-assignment2-bufferandstruct-DSnoNintendo$
```

Screen shot(s) of the execution of the program:



```
student@student:~/csc415-assignment2-bufferandstruct-DSnoNintendo$ make run
./Stronge_Daryl_HW2_main Daryl Stronge "Four score and seven years ago our fathers brought fort
h on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that
all men are created equal."
------------------------------------ CHECK ------------------------------------
Running the check for Daryl Stronge
Name check is 0 by 0
Student ID: 917844673, Grade Level: Senior
Languages: 262162
Message:
Four score and seven years ago our fathers brought forth on this continent, a new nation, conce
ived

The Check Succeded (0, 0)

END-OF-ASSIGNMENT
000000: 30 13 AF AC B1 C9 00 00  50 13 AF AC B1 C9 00 00 | 0.????..P.????..
000010: C1 32 B5 36 14 00 00 00  12 00 04 00 46 6F 75 72 | ?2?6........Four
000020: 20 73 63 6F 72 65 20 61  6E 64 20 73 65 76 65 6E |  score and seven
000030: 20 79 65 61 72 73 20 61  67 6F 20 6F 75 72 20 66 |  years ago our f
000040: 61 74 68 65 72 73 20 62  72 6F 75 67 68 74 20 66 | athers brought f
000050: 6F 72 74 68 20 6F 6E 20  74 68 69 73 20 63 6F 6E | orth on this con
000060: 74 69 6E 65 6E 74 2C 20  61 20 6E 65 77 20 6E 61 | tinent, a new na
000070: 74 69 6F 6E 2C 20 63 6F  6E 63 65 69 76 65 64 20 | tion, conceived

student@student:~/csc415-assignment2-bufferandstruct-DSnoNintendo$
```