Name: Shorena K. Anzhilov
Email: anzhilov.s@northeastern.edu
Spring, 2025

Question 1a: Stack and Heap Collision

When the stack and heap grow towards each other and eventually meet in the middle of the memory space, it causes serious issues for the program. This situation is known as a stack overflow or heap corruption. Since both the stack and heap are trying to use the same memory space, the program may crash because it runs out of available memory. Another issue is that data from the stack might overwrite data in the heap, or vice versa, leading to unpredictable program behavior and incorrect results. In some cases, this can also create security vulnerabilities, as critical memory regions might get overwritten. Additionally, if the memory is exhausted, operations like function calls or memory allocations may fail, causing the program to behave unexpectedly or terminate. To prevent this, programmers need to makes sure of efficient memory management and check for available space before allocating new memory on the heap or pushing data onto the stack.

Question 1b: System Call Table Benefits

Older computer systems without memory protection allowed programs to call operating system functions directly using fixed memory addresses. However, using a system call table instead of direct calls provides several benefits. One important reason is portability, if the operating system changes, the addresses of functions might also change, which could break programs that use direct calls. A system call table provides a stable interface, allowing programs to work even if the OS updates. Another reason is consistency, the call table makes sure that programs always access the correct OS functions in a structured way. Additionally, the call table allows error handling and updates without affecting the programs using it. Security is another key reason for using a call table, as it helps control access to critical system functions, preventing accidental or malicious misuse. If a program directly calls OS functions and the addresses change, it may crash, while a program using the system call table will continue to work reliably.

Question 1c: Issues in Given C Code

The given C code in Question 1c contains a few issues that could cause the program to fail. First, the `calloc()` function is used incorrectly. It requires two arguments: the number of elements and the size of each element, but the code only provides one argument. This can be fixed by using `calloc(1011, sizeof(char))` to allocate the correct amount of memory. Second, the `memset()` function is incorrectly applied. The code uses `&block,` which takes the address of the pointer instead of the allocated memory. The correct way is to use `memset(block, 0xFF, 1011);` to fill the allocated memory correctly. Another issue is the lack of error checking after memory allocation. If `calloc()` fails and returns NULL, the program may crash when trying to use the memory. To fix this, an `if` condition should be added to check if memory allocation was successful before proceeding. These changes makes sure that the program allocates and initializes memory properly without causing unexpected errors.

```
int main() {
   // Allocate memory for 1011 bytes
   char* block = calloc(1011, sizeof(char));

   // Check if memory allocation was successful
   if (block == NULL) {
      printf("Memory allocation failed\n");
  // Exit with an error code
      return 1;
   }

   // Fill the allocated memory with 0xFF (255 in decimal)
   memset(block, 0xFF, 1011);

   // Example usage: print the first few values to verify
   for (int i = 0; i < 10; i++) {
      printf("block[%d] = %X\n", i, (unsigned char)block[i]);
   }

   // Free the allocated memory to avoid memory leaks
   free(block);
   printf("Memory successfully freed.\n");

  // Successful execution
   return 0;
}
```

This corrected version properly allocates and initializes memory while handling potential errors.