Question 1:

When the stack and heap grow towards each other and meet, a stack-heap collision or overflow occurs.

If the stack overextends into the heap space, the runtime stack operations can overwrite the dynamically allocated memory on the heap. This overwriting can lead to data corruption, where data is written to incorrect memory locations, potentially causing data loss, security vulnerabilities, or program crashes.

Conversely, if the heap expands into the stack space, memory allocation functions such as malloc in C will fail to allocate more memory. These functions will return a null pointer. If the program does not properly check for these failure signals and handle them, it may crash. The OS’s memory manager enforces memory access rights, and any illegal access, such as attempting to use memory beyond the allowed space, will result in a segmentation fault or access violation, causing the program to terminate.

Question 2:

Firstly, direct function calls to the operating system (OS) pose a significant risk when the system undergoes an update or change. If an application calls an OS function at a specific memory address and the OS is later updated such that the function is moved to a different address, the direct call will fail. This is because the call attempts to execute code at the previous address, which no longer corresponds to the intended function due to the update, potentially leading to application crashes.

On the other hand, if a system call table is used, it employs identifiers for functions instead of fixed memory addresses. This design ensures that when the OS is updated and functions are relocated, the system call table can be revised to reflect the new addresses. Applications invoking functions through this table can continue to operate correctly, as they are automatically redirected to the new addresses without any need to alter the application code itself.

Question 3:

The first issue with the provided code is that the calloc function requires two arguments: the number of elements and the size of each element. The correct usage is calloc(2048, sizeof(char))

Second problem, the memset function passes the address of the block but not the pointer of block itself. It will not set the allocated block of memory to 0xFF but will instead overwrite the pointer. The declaration of memset() function should be void \*memset(void \*str, int c, size\_t n). As you can see, str is a pointer to the block of memory but not the address. So in the code, it writes as &block which will actually overwrites the memory where “block” is stored but will set the pointer address of the “block” to the new address 0xFF. The correct usage of memset is to pass it a pointer to the block of memory to fill, the value to fill it with, and the number of bytes to fill.

Also, at the end, it has no free call to release the memory allocated by calloc.

One more thing, it did not check if block is NULL or not.

The code should be like this:

char\* block = calloc(2048, sizeof(char));

if (block == NULL) {

printf(“block is null”);

}

Memset(block, 0xFF, 2048);

free(block);