# CS5600: Computer Systems

Week 4: Assignment / Memory Layout and Memory Management February 2, 2025

**Student:** Ben Henshaw, henshaw.b@northeastern.edu

**Problem 1: Memory Layout**

Stacks on almost all modern CPUs grow down, using lower memory addresses as more data is pushed onto the stack. The heap refers to memory allocated by `malloc()` or the C++ or Java `new` operator. A typical heap allocator will start with a certain amount of memory and allocate more if that runs out.

(a) If memory is laid out in this manner, what happens when the stack and the heap meet in the middle? Explain, in your own words, what would occur and what the consequences of that event would be on the program.

(b) For single-user operating systems without memory protection (e.g., the CSx600, Z80 or Intel 808x) applications are able to make direct function calls to operating system code, yet most operating systems which ran on such computers (e.g., MS-DOS, early MacOS) provided a system call table for making indirect calls into the operating system, and these tables were almost universally used by applications. Describe a reason for using such a table, i.e., a case in which a program calling operating system addresses directly might fail in some cases, while one using the system call table would not.

(c) What is wrong with the following C code fragment? Describe the issue or issues and how you would need to modify to code to avoid any potential problems.

```
char* block = calloc(2048);
memset(&block, 0xFF, 2048);
```

(a) The event in which data are pushed to the stack until the structure encroaches on the space allocated to heap memory, and the event in which the same happens as heap allocations encroach the space for stack memory are known as **stack overflow** and **heap overflow** overflow respectively. Both are types of a **buffer overflow** in which the memory value is

The **call stack** is a data structure in which a system stores the local variables and return addresses that are utilized in the scope of active processes. Large processes with considerable overhead (such as those that utilize deep recursion or have large automatic variables) can cause the call stack to expand beyond the stack's bounds. If it does so, data at memory locations allocated for dynamic variables may now reference data that has been pushed to the stack. If stack growth continues unchecked, it may start to overwrite the heap, before continuing to overwrite any static memory allocated for the program. University of Alberta (2000)

The **heap** is a memory location for dynamically-allocated data that is assigned at runtime.

Dynamically-allocated memory needs to be freed when it is no longer being used. If memory isn't freed as new memory is being allocated, it's possible to cause memory leaks where the bottom of the stack may be randomly overwritten by newly-allocated data by exceeding the stack pointer.

As anyone who has experimented with recursion in `C` knows, uncontrolled recursion of this type often results in the program crashing due to a `segmentation fault`. This occurs because of the way modern memory management works, specifically how the process of **segmentation** is used to protect memory access. Simplilearn (2024) This concept involves reserving sections of memory in *segments* for specific applications. In this case, a segment would be allocated for the call stack associated with a process or with the OS at large, and similarly a segment would be defined for the heap. Modern CPUs are capable of executing **interrupts**, (also known as **trap** instructions) that accordingly interrupt the execution of a program. zwol (2016) When the bounds of a segment are violated, the kernel sends a `SIGSEV` signal to the processor, which signals for the process to be terminated in an uncontrolled crash using an interrupt. Dancuk (2023) Theoretically, this should prevent out-of-bounds memory access within the stack and thus averts undefined behavior that may result, causing the memory for the process to be deallocated.

(b) One of the first goals of an operating system is to provide an abstract interface with the hardware, and in order to be useful and usable on multiple platforms, the way it interacts with the programs that are written to it must be standardized in some way. For programs written for that OS, there is a table of system calls that can be referenced by programs to make subroutine calls that allow it to interact with the hardware. A **call table** is such an interface located at some fixed point in memory so that any programs written for the OS in question know where to seek out those instructions. Without a dedicated call table, the location of those instructions do not have a standardized place, and thus a program for that system has to be rewritten each time it is adapted to a similar platform, as the location of those calls would become platform- or hardware-dependent.

(c) The first line of code allocates 2048 bytes of space, or 2 kb, from dynamic memory for a `char` pointer. One issue with this line is that—although a char is appropriately 1 byte in size and that 2048 bytes provides enough space for 2048 characters–the amount of space allocated is not expressed in terms of `sizeof(char)`, which is a good practice to ensure there is enough memory allocated to accommodate a specific amount fo those values. It is also good practice to coerce the type of the return from the `malloc()` call, as it by default returns a void pointer, and assigning values to that pointer may result in undefined behavior, such as how `int` and `char` are both represented similarly under the hood, but their actual behavior is context dependent.

The call to `memset()` assigns the value `0xFF` corresponding to the ASCII character 'U' to each byte starting at the memory address indicated by `block`. This is similar to declaring a string of equivalent length, but the problem is that it does not assign a string terminator to any point in the block of memory. Any functions that would treat a character array as a string would not know where it is intended to end, and may read garbage data past the memory location `block + 2048`. This would be an example of undefined behavior.

## References

Dancuk, M. (2023). *What is Signal 11 SIGSEGV Error?* https://phoenixnap.com/kb/sigsegv.

Simplilearn (2024). *The Ultimate Guide to Segmentation in Operating Systems*. https://www.simplilearn.com/segmentation-in-os-article.

University of Alberta, C. (2000). *Using and Porting GNU Fortran*. https://web.archive.org/web/20120906102106/http://sunsite.ualberta.ca/Documentation/Gnu/gcc-2.95.2/html_node/g77_toc.html.

zwol (2016). *How does a Segmentation Fault work under-the-hood?* https://unix.stackexchange.com/questions/257598/how-does-a-segmentation-fault-work-under-the-hood.