Lecture Notes on Memory Management

15-122: Principles of Imperative Computation
Frank Pfenning, Rob Simmons, Andrè Platzer

Lecture 11
June 4, 2014

Ultimately, "all" data resides in memory. In fact, part of the data may also be kept in fast registers directly on the CPU. You will learn about registers in detail in 15-213 and can learn about their use in programming languages in 15-411. For the purposes of today's lecture, it is sufficient to pretend all data would sit in memory and ignore registers for the time being. This simplifies the principles without losing too much precision.

The data in memory is addressed by memory addresses that fit to the addressing of the CPU in your computer. We will just pretend 32bit addresses, because those are shorter to write down. All addresses are positive, so the lowest address is 0x00000000 and the highest address $2^{32} - 1 =$ `0xFFFFFFFF`. All data (with the caveat about registers) sits in memory at some address. One important question about all data in memory is how big it is, so that the compiler can make sure program data is stored without accidental overlapping regions.

The basic memory layout looks as follows:

```
OS AREA
============
System stack   (local variables and function calls)
============
unused
============
System heap    (data allocated here... alloc or alloc_array)
============
.text (read only)  (program instructions sit in memory)
============
OS AREA
```

One consequence of this memory layout is that the stack grows towards the heap, and the heap usually grows towards the stack. The reason that the stack is called a stack is because it operates somewhat like the principle of the stack data structure. Your program can put new data on the top of the stack. It can also pop elements of the stack if this data is no longer necessary. Unlike the stack abstraction, it may appear as if your program internally also modifies data that is on the stack, even if it is not quite at the top of it. However, the only data on the stack that the program modifies is in the top range of the stack (perhaps the top 512 bytes or so, depending on the function that runs), even if it is not just the top word of the stack.

Programs cannot access memory cells that belong to the operating system. If they try, programs get an "exception" like a segmentation fault. Where can that happen in C0? C0 takes great care to ensure that it never gives you any pointers to uninitialized or random or garbage data in memory, *except*, of course, the `NULL` pointer. `NULL` is a special pointer to the memory address 0, which belongs to the operating system. Any access by a user-land program by dereferencing a NULL pointer causes a segfault.

If, however, you are writing a program that will be running as part of the operating system, your program has no protection against `NULL` pointer dereferencing anymore, because memory address 0 is a valid address for the operating system, even if not for regular programs. When writing code for operating systems, you, thus, need to have mastered the art of protecting against illegal `NULL` dereferences. This is exactly one of the things that contracts, loop invariants, and assertions prepare you for.