CMPSC 311 Scribe Notes

Address Space, Stack, and Heap

1/22/16 Lecture 5

Overview

Previously we had discussed the address space of a program. If you've forgotten, recall that an address space can be defined as a sequence of contiguous virtual addresses (multiple of bytes) that a process is provided by the operating system with the guarantee that it will be completely isolated from the actions of other processes.

The four main components of an address space are:

1. The Code Segment
2. The Global (or Static) Variables
3. The Stack
4. The Heap

Today, we're going to discuss both the stack and the heap.

The Stack

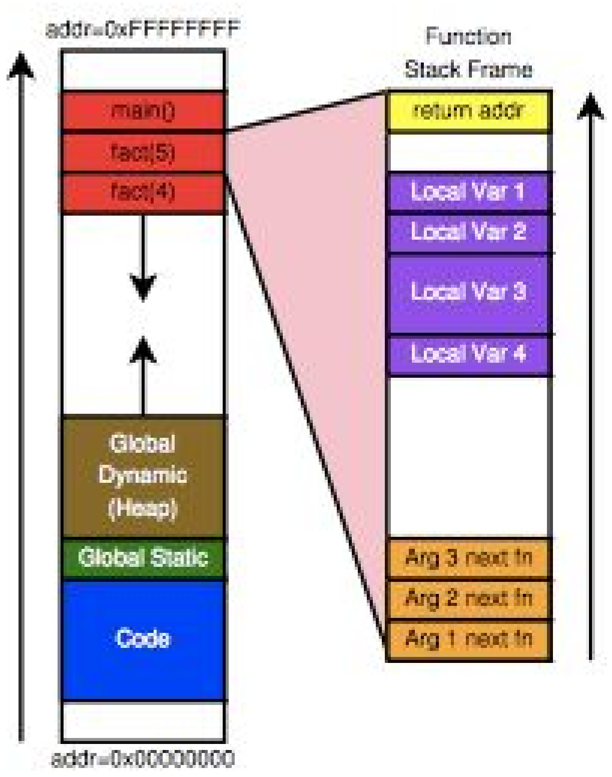
The ***call stack***, or sometimes just the stack, is responsible for storing data relevant to a function when said function is invoked. This includes:

- local variables
- temporary variables
- arguments for function calls
- return address

" **Note:** The stack is a data structure that evolves dynamically as the process run. *"*

The exact meanings of these are outlined below, but first understand that all of the above data is stored within something called a *stack frame* or simply ***frame***. Each function is dynamically allocated its own frame with which to store the necessary data. It's important to understand that a frame is created every time a function is called, and it is destroyed (*deallocated*) every time a function completes.

Local variables: these, as you may have learned in previous courses, are variables limited to just the scope of the function they are contained within.
Temporary variables: these variables are not explicitly part of your written program. The compiler creates these behind the scenes in order to keep track of the internal state of your function call. See the next page for an example.
Arguments for function calls: if your function, A, calls another function, B, with the argument 10, then the program must temporarily store the 10 argument and pass it along to the next function frame.
Return address: after a function finishes, it needs to know where in the code to return to. The return address indicates that location in the code segment of the address space



On the left, you can see the stack in red. Each rectangular slice represents a *stack frame*. The top frame of any C program will always be `main()` and subsequent frames are added beneath the previous one.

Similarly to the data structure of the same name, the stack, follows the last-in-first-out (*LIFO*) principle. In this case, the last frame placed onto the stack will be the first frame removed, or *popped* from the stack, as the last called function will be the first to complete.

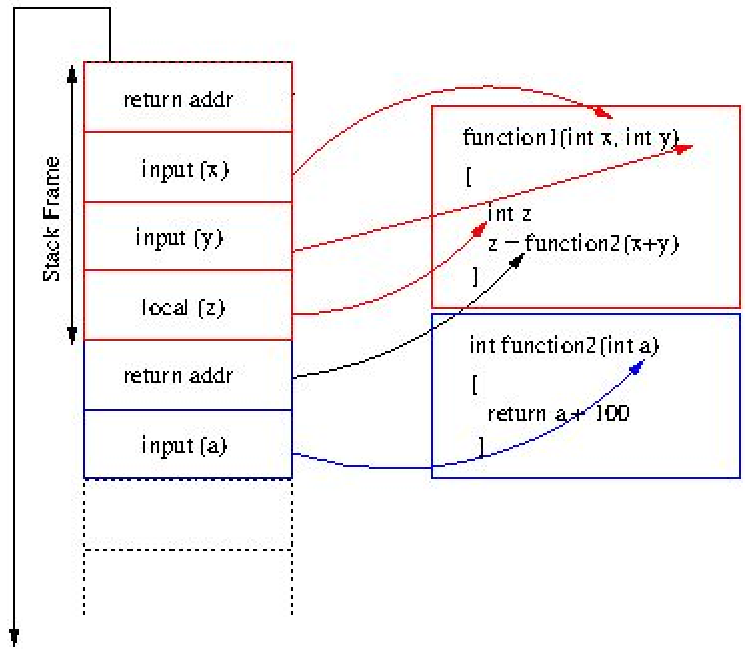
" **Note:** `main()` stays on stack until the end of the program. "

Scribe Question: Where do you think `main()` returns? This will be answered later in the course.

Stack Example

To the right is an illustration on how function calls work within the context of the stack.

Ignoring the existence of `main()`, the first function called, *function1* (the red one), is placed on the top of the stack with its corresponding metadata. When *function2* (the blue one) is reached within *function1*, its stack frame is then placed upon the stack.



Source: http://www.bottomupcs.com/elements_of_a_process.xhtml

The Heap

Not to be confused with the data structure of the same name (no relation), the *heap* is part of the address space used for longer-term storage that persists across function calls. What distinguishes this from the global variables segment within the address space is that the heap is for dynamically allocated variables, which are not necessarily global.

If you're familiar with C++, the `new` and `delete` keywords are analogous to C's `malloc` and `free` keywords. Both methods are located within the standard library `<stdlib.h>`.

Malloc

The `malloc()` function allocates *`size`* bytes and returns a pointer to the allocated memory.

```
void * malloc(size_t size);
```

If *size* is 0, then `malloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

Free

The `free()` function deallocates (i.e, frees) the memory space pointed to by *`ptr`*, which must have been returned by a previous call to `malloc()`.

```
void free(void *ptr);
```

Notice that `malloc()` returns a void pointer, which can be cast into any pointer type, and `free()` does not return anything.

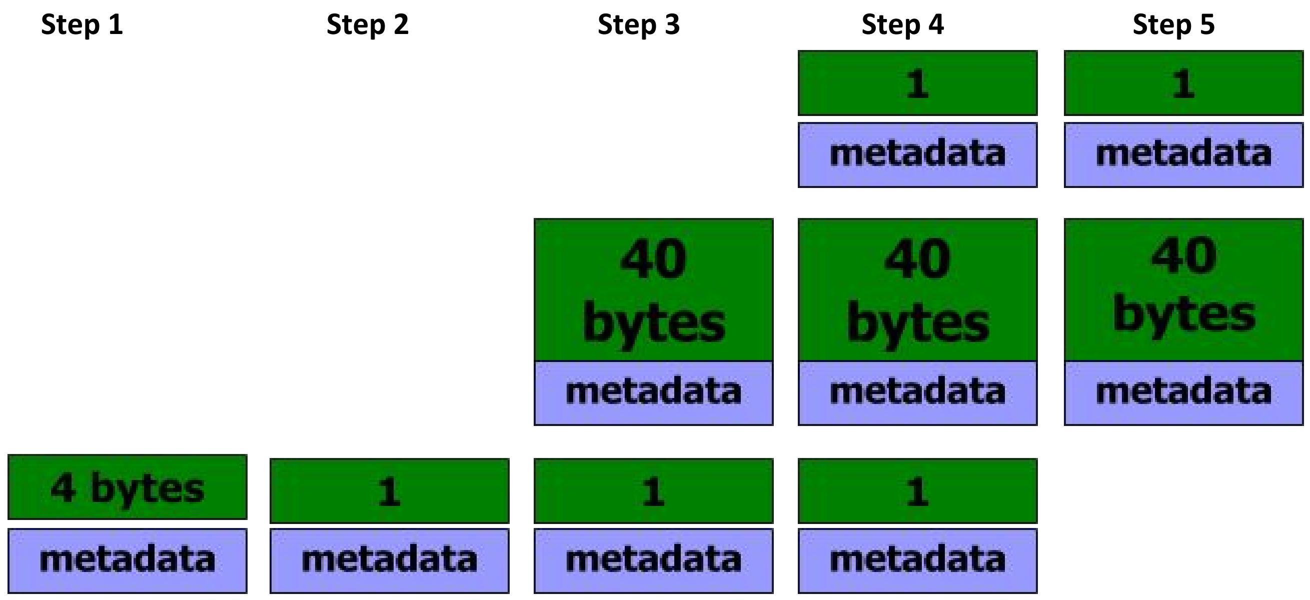
Heap Example

Consider the following C code below:

```
#include <stdio.h>
#include <stdlib.h>

int main () {
      int *p, *q, *r;
      p = (int *) malloc (sizeof(int));         // step 1
      *p = 1;                                   // step 2
      q = (int *) malloc (10*sizeof(int));      // step 3
      r = (int *) malloc (sizeof(int));         // step 4
      free (p);                                 // step 5
      return 0;
}
```

When we reach the first `malloc`, the pointer `p` is assigned the address on the heap meant for an integer (4 bytes). This marks the beginning of the labeled steps and can be visually depicted as seen on the next page. The comments in the above code correspond to the visual steps.



Notice that as long as `free` is not called, the memory will remain allocated. It's therefore possible to have *memory leaks*, or data that will never be used again yet still remains in memory. It's also possible to run out of memory as the address space is finite.

" **Note:** heap reduces the amount of memory to be used. It is more efficient than declaring variables in the code. "