Memory management in C: The heap and the stack

Leo Ferres
Department of Computer Science
Universidad de Concepción
leo@inf.udec.cl

October 7, 2010

1 Introduction

When a program is loaded into memory, it's organized into three areas of memory, called *segments*: the *text* segment, the *stack* segment, and the *heap* segment. The text segment (sometimes also called the code segment) is where the compiled code of the program itself resides. This is the machine language representation of the program steps to be carried out, including all functions making up the program, both user and system defined[1]. Thus, in general, an executable program generated by a compiler (like gcc) will have the following organization in memory on a typical architecture (such as on MIPS)[2]:

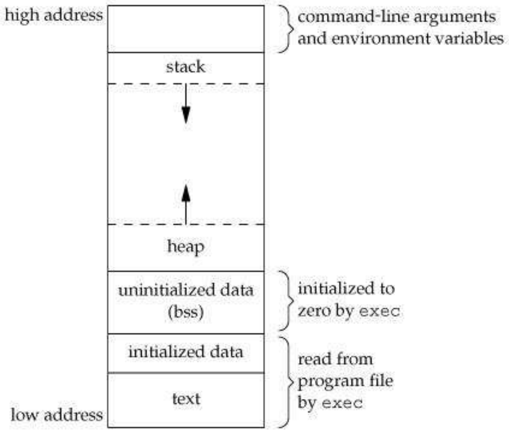
high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Figure 1: Memory organization of a typical program in MIPS

where[3]

- Code segment or text segment: Code segment contains the code executable or code binary.
- Data segment: Data segment is sub divided into two parts
  - Initialized data segment: All the global, static and constant data are stored in the data segment.
  - Uninitialized data segment: All the uninitialized data are stored in BSS.

[1] http://ee.hawaii.edu/~tep/EE160/Book/chap14/subsection2.1.1.8.html

[2] http://129.107.52.7/cse5317/notes/node33.html

[3] http://www.boundscheck.com/knowledge-base/c-cpp/memory-layout-in-c/342/

- Heap: When program allocate memory at runtime using calloc and malloc function, then memory gets allocated in heap. when some more memory need to be allocated using calloc and malloc function, heap grows upward as shown in above diagram.

- Stack: Stack is used to store your local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over. When a new stack frame needs to be added (as a result of a newly called function), the stack grows downward.

The stack and heap are traditionally located at opposite ends of the process's virtual address space. The stack grows automatically when accessed, up to a size set by the kernel (which can be adjusted with setrlimit(`RLIMIT_STACK`, ...)). The heap grows when the memory allocator invokes the `brk()` or `sbrk()` system call, mapping more pages of physical memory into the process's virtual address space.

Implementation of both the stack and heap is usually down to the runtime/OS. Often games and other applications that are performance critical create their own memory solutions that grab a large chunk of memory from the heap and then dish it out internally to avoid relying on the OS for memory.

2 Stack

Stacks in computing architectures are regions of memory where data is added or removed in a last-in-first-out manner[4]. In most modern computer systems, each thread has a reserved region of memory referred to as its stack. When a function executes, it may add some of its state data to the top of the stack; when the function exits it is responsible for removing that data from the stack. At a minimum, a thread's stack is used to store the location of function calls in order to allow return statements to return to the correct location, but programmers may further choose to explicitly use the stack. If a region of memory lies on the thread's stack, that memory is said to have been allocated on the stack (see Footnote 4).

Because the data is added and removed in a last-in-first-out manner, stack allocation is very simple and typically faster than heap-based memory allocation (also known as dynamic memory allocation). Another feature is that memory on the stack is automatically, and very efficiently, reclaimed when the function exits, which can be convenient for the programmer if the data is no longer required. If however, the data needs to be kept in some form, then it must be copied from the stack before the function exits. Therefore, stack based allocation is suitable for temporary data or data which is no longer required after the creating function exits (see Footnote 4).

A call stack is composed of stack frames (sometimes called activation records). These are machine dependent data structures containing subroutine state information. Each stack frame corresponds to a call to a subroutine which has not yet terminated with a return. For example, if a subroutine named DrawLine is currently running, having just been called by a subroutine DrawSquare, the top part of the call stack might be laid out like this (where the stack is growing towards the top)[5]:

Here are a few additional pieces of information about the stack that should be mentioned[6]:

- The OS allocates the stack for each system-level thread when the thread is created. Typically the OS is called by the language runtime to allocate the heap for the application.

- The stack is attached to a thread, so when the thread exits the stack is reclaimed. The heap is typically allocated at application startup by the runtime, and is reclaimed when the application (technically process) exits.

- The size of the stack is set when a thread is created.

[4] `http://en.wikipedia.org/wiki/Stack-based_memory_allocation`

[5] `http://en.wikipedia.org/wiki/Call_stack`

[6] `http://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap/80113#80113`

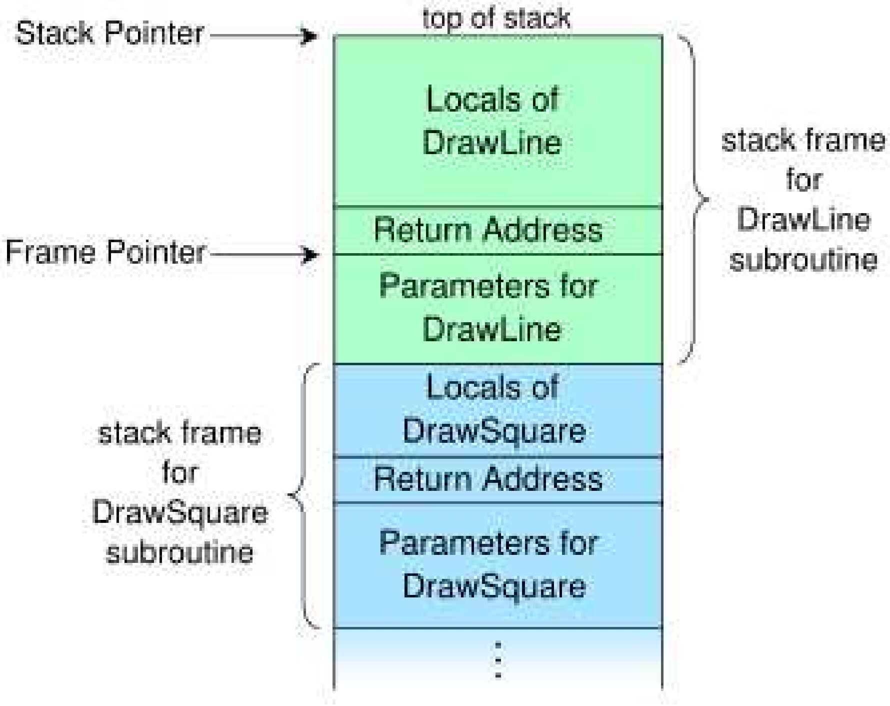

Figure 2: Stack frame for some DrawRectangle program

- The stack is faster because the access pattern makes it trivial to allocate memory from it, while the heap has much more complex bookkeeping involved in an allocation or free. Also, each byte in the stack tends to be reused very frequently which means it tends to be mapped to the processor's cache, making it very fast.
- Stored in computer RAM like the heap.
- Variables created on the stack will go out of scope and automatically deallocate.
- Much faster to allocate in comparison to variables on the heap.
- Implemented with an actual stack data structure.
- Stores local data, return addresses, used for parameter passing
- Can have a stack overflow when too much of the stack is used. (mostly from inifinite (or too much) recursion, very large allocations)
- Data created on the stack can be used without pointers.
- You would use the stack if you know exactly how much data you need to allocate before compile time and it is not too big.
- Usually has a maximum size already determined when your program starts.

Some implementation examples are the following:

- In C you can get the benefit of variable length allocation through the use of alloca, which allocates on the stack, as opposed to alloc, which allocates on the heap. This memory won't survive your return statement, but it's useful for a scratch buffer.

Deallocating the stack is pretty simple because you always deallocate in the reverse order in which you allocate. Stack stuff is added as you enter functions, the corresponding data is removed as you exit them. This means that you tend to stay within a small region of the stack unless you call lots of functions that call lots of other functions (or create a recursive solution)[7].

2.1 Stack overflow

That said, stack-based memory errors are some of the worst I've experienced. If you use heap memory, and you overstep the bounds of your allocated block, you have a decent chance of triggering a segment fault. (Not 100%: your block may be incidentally contiguous with another that you have previously allocated.) But since variables created on the stack are always contiguous with each other, writing out of bounds can change the value of another variable. I have learned that whenever I feel that my program has stopped obeying the laws of logic, it is probably buffer overflow[8].

Let me show you some examples of ways to kill the stack:

```
int main(){
  main();
}
```

or the similar

```
int add(int n)
{
  return n + add(n + 1);
}
```

whereby we blow the stack by simply using more memory than is available to it for this thread.

The other common error to be careful with is buffer overflow, that is, when we write past the end of some variable, overwriting vital information. Take the following sample code:

```
#include <string.h>
void foo (char *bar)
{
  char  c[12];
  strcpy(c, bar);  // no bounds checking...
}

int main (int argc, char **argv)
{
  foo(argv[1]);
}
```

Notice in figure C above, when an argument larger than 11 bytes is supplied on the command line foo() overwrites local stack data, the saved frame pointer, and most importantly, the return address. When foo() returns it pops the return address off the stack and jumps to that address (i.e. starts executing instructions from that address). As you can see in figure C above, the attacker has overwritten the return address with a pointer to the stack buffer char c[12], which now contains attacker supplied data. In an actual stack buffer overflow exploit the string of "A"'s would be replaced with shellcode suitable to the platform and desired

[7] http://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap/79988#79988

[8] http://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap/662783#662783

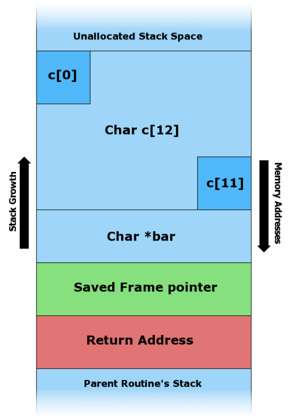

Figure 3: Before data is copied.

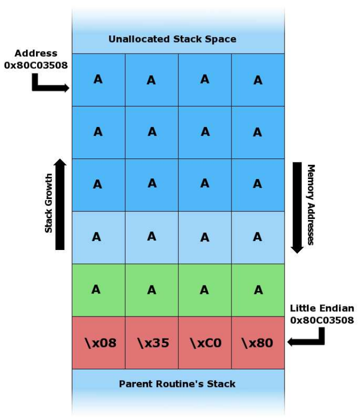

Figure 4: After data ¿ 12 chars is copied.

function. If this program had special privileges (e.g. the SUID bit set to run as the superuser), then the attacker could use this vulnerability to gain superuser privileges on the affected machine[9].
or the similar

```
char* foo()
{
    char str[256];
    return str;
}

void bar()
{
    char* str = foo();
    strcpy(str, "Holy sweet Moses! I blew my stack!!");
}
```

3 Heap

The heap contains a linked list of used and free blocks. New allocations on the heap (by new or malloc) are satisfied by creating a suitable block from one of the free blocks. This requires updating list of blocks on the heap. This meta information about the blocks on the heap is also stored on the heap often in a small area just in front of every block [10].

- The size of the heap is set on application startup, but can grow as space is needed (the allocator requests more memory from the operating system) (see Footnote 6).
- Stored in computer RAM like the stack.
- Variables on the heap must be destroyed manually and never fall out of scope. The data is freed with delete, delete[] or free
- Slower to allocate in comparison to variables on the stack.
- Used on demand to allocate a block of data for use by the program.

[9] http://en.wikipedia.org/wiki/Stack_buffer_overflow

[10] http://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap/1213360#1213360

- Can have fragmentation when there are a lot of allocations and deallocations
- Can have allocation failures if too big of a buffer is requested to be allocated.
- You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.
- Responsible for memory leaks

So consider a run of a program[11]

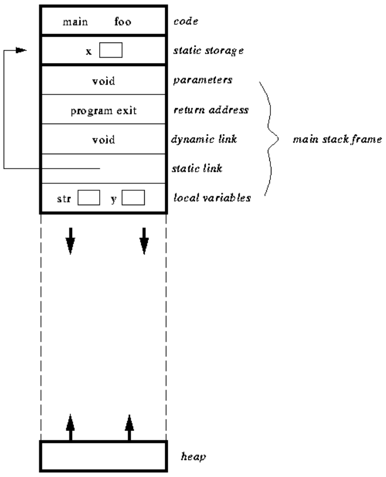

Figure 5: At start of the program

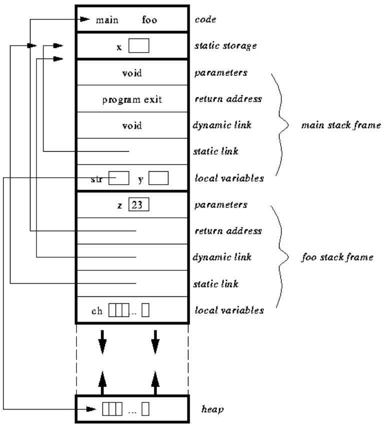

Figure 6: At first call for foo

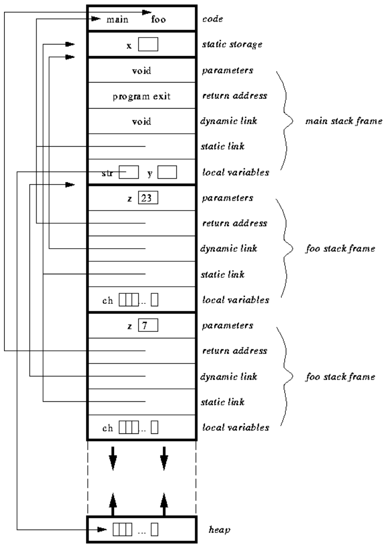

Figure 7: After second call to Foo

```
int x;                /* static storage */

void main() {
   int y;             /* dynamic stack storage */
   char *str;         /* dynamic stack storage */

   str = malloc(100); /* allocates 100 bytes of dynamic heap storage */

   y = foo(23);
   free(str);         /* deallocates 100 bytes of dynamic heap storage */
}                     /* y and str deallocated as stack frame is popped */

int foo(int z) {      /* z is dynamic stack storage */
   char ch[100];     /* ch is dynamic stack storage */

   if (z == 23) foo(7);

   return 3;          /* z and ch are deallocated as stack frame is popped,
                         3 put on top of stack  */
}
```

[11] http://www.cs.jcu.edu.au/Subjects/cp2003/1997/foils/heapAndStack/heapAndStack.html

3.1 Memory leaks

A memory leak, in computer science (in such context, it's also known as leakage), occurs when a computer program consumes memory but is unable to release it back to the operating system. A memory leak has symptoms similar to a number of other problems and generally can only be diagnosed by a programmer with access to the program source code; however, many people refer to any unwanted increase in memory usage as a memory leak, though this is not strictly accurate (Info here is in [12]).

A memory leak can diminish the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down unacceptably due to thrashing.

Memory leaks may not be serious or even detectable by normal means. In modern operating systems, normal memory used by an application is released when the application terminates. This means that a memory leak in a program that only runs for a short time may not be noticed and is rarely serious.

Typically, a memory leak occurs because dynamically allocated memory has become unreachable. The prevalence of memory leak bugs has led to the development of a number of debugging tools to detect unreachable memory. IBM Rational Purify, BoundsChecker, Valgrind, Insure++ and memwatch are some of the more popular memory debuggers for C and C++ programs. "Conservative" garbage collection capabilities can be added to any programming language that lacks it as a built-in feature, and libraries for doing this are available for C and C++ programs. A conservative collector finds and reclaims most, but not all, unreachable memory.

The following C function deliberately leaks memory by losing the pointer to the allocated memory. Since the program loops forever calling the memory allocation function, malloc(), but without saving the address, it will eventually fail (returning NULL) when no more memory is available to the program. Because the address of each allocation is not stored, it is impossible to free any of the previously allocated blocks. It should be noted that, generally, the operating system delays real memory allocation until something is written into it. So the program ends when virtual addresses run out of bounds (per process limits or 2 to 4 GiB on IA-32 or a lot more on x86-64 systems) and there may be no real impact on the rest of the system.

```
#include <stdlib.h>

int main(void)
{
     /* this is an infinite loop calling the malloc function which
      * allocates the memory but without saving the address of the
      * allocated place */
     while (malloc(50)); /* malloc will return NULL sooner or later, due to lack of memory */
     return 0;  /* free the allocated memory by operating system itself after program exits */
}
```

And here's another example, following the same idea.

```
#include <stdlib.h>
void f(void)
 {
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;        // problem 1: heap block overrun
 }                    // problem 2: memory leak -- x not freed
  int main(void)
 {
   f();
   return 0;
 }
```

[12] http://en.wikipedia.org/wiki/Memory_leak