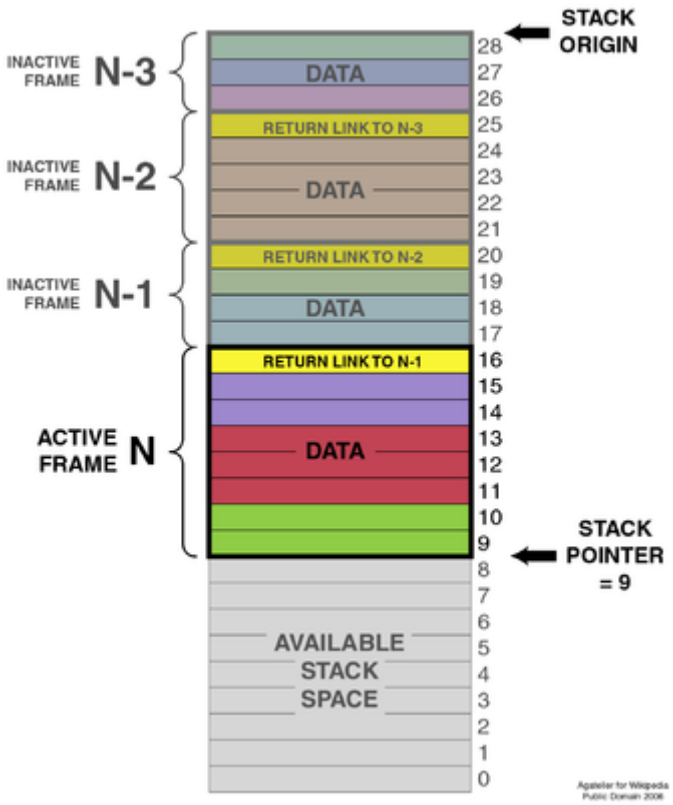
Stack-based memory allocation

Stacks in computing architectures are regions of memory where data is added or removed in a last-in-first-out (LIFO) manner.

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 local 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 a return address provided by the caller in order to allow return statements to return to the correct location. The stack is often used to store variables of fixed length local to the currently active functions. Programmers may further choose to explicitly use the stack to store local data of variable length. If a region of memory lies on the thread's stack, that memory is said to have been allocated on the stack, i.e. **stack-based memory allocation**.



A typical stack, storing local data and call information for nested procedure calls (not necessarily nested procedures). This stack grows downward from its origin. The stack pointer points to the current topmost datum on the stack. A push operation decrements the pointer and copies the data to the stack; a pop operation copies data from the stack and then increments the pointer. Each procedure called in the program stores procedure return information (in yellow) and local data (in other colors) by pushing them onto the stack.

Contents

Advantages and disadvantages

Because the data is added and removed in a last-in-first-out manner, stack-based memory allocation is very simple and typically much faster than heap-based memory allocation (also known as dynamic memory allocation) typically allocated via malloc. 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.[1] (The same applies to longjmp if it moved to a point before the call to `alloca` happened.) If however, the data needs to be kept in some form, then it must be copied from the stack to the heap before the function exits. Therefore, stack based allocation is suitable for temporary data or data which is no longer required after the current function exits.

A thread's assigned stack size can be as small as only a few bytes on some small CPUs. Allocating more memory on the stack than is available can result in a crash due to stack overflow. This is also why functions that use `alloca` are usually prevented from being inlined:[2] should such a function be inlined into a loop,

the caller would suffer from an unanticipated growth in stack usage, making an overflow much more likely.

Stack-based allocation can also cause minor performance problems: it leads to variable-size stack frames, so that both stack and frame pointers need to be managed (with fixed-size stack frames, one of these is redundant). This is usually much less costly than calling `malloc` and `free` anyway. In particular, if the current function contains both calls to alloca and blocks containing variable length local data then a conflict occurs between alloca's attempts to increase the current stack frame until the current function exits versus the compiler's need to place local variables of variable length in the same location in the stack frame. This conflict is typically resolved by creating a separate chain of heap storage for each call to alloca (see: https://code.woboq.org/gcc/libiberty/alloca.c.html). The chain records the stack depth at which each allocation occurs, subsequent calls to alloca in any function trim this chain down to the current stack depth to eventually (but not immediately) free any storage on this chain. A call to alloca with an argument of zero can also be used to trigger the freeing of memory without allocating any more such memory. As a consequence of this conflict between alloca and local variable storage, using alloca might be no more efficient than using malloc.

System interface

Many Unix-like systems as well as Microsoft Windows implement a function called `alloca` for dynamically allocating stack memory in a way similar to the heap-based `malloc`. A compiler typically translates it to inlined instructions manipulating the stack pointer, similar to how variable-length arrays are handled.[3] Although there is no need to explicitly free the memory, there is a risk of undefined behavior due to stack overflow.[4] The function was present on Unix systems as early as 32/V (1978), but is not part of Standard C or any POSIX standard.

A safer version of `alloca` called `_malloca`, which reports errors, exists on Microsoft Windows. It requires the use of `_freea`.[5] gnulib provides an equivalent interface, albeit instead of throwing an SEH exception on overflow, it delegates to `malloc` when an overlarge size is detected.[6] A similar feature can be emulated using manual accounting and size-checking, such as in the uses of `alloca_account` in glibc.[7]

Some processor families, such as the x86, have special instructions for manipulating the stack of the currently executing thread. Other processor families, including PowerPC and MIPS, do not have explicit stack support, but instead rely on convention and delegate stack management to the operating system's application binary interface (ABI).

References

1. "Advantages of Alloca" (https://www.gnu.org/software/libc/manual/html_node/Advantages-of-Alloca.html). *The GNU C Library*.
2. "Inline" (http://gcc.gnu.org/onlinedocs/gcc/Inline.html). *Using the GNU Compiler Collection (GCC)*.
3. `alloca(3) (http://man7.org/linux/man-pages/man3/alloca.3.html)` – Linux Programmer's Manual – Library Functions
4. "Why is the use of alloca() not considered good practice?" (https://stackoverflow.com/questions/1018853/why-is-the-use-of-alloca-not-considered-good-practice). *stackoverflow.com*. Retrieved 2016-01-05.
5. "_malloca" (https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/malloca?view=vs-2019). *Microsoft CRT Documentation*.
6. "gnulib/malloca.h" (https://github.com/coreutils/gnulib/blob/master/lib/malloca.h). *GitHub*. Retrieved 24 November 2019.

7. "glibc/include/alloca.h" (https://github.com/bminor/glibc/blob/780684eb04298977bc411ebca1eadeeba4877833/include/alloca.h). Beren Minor's Mirrors. 23 November 2019.

See also

- Automatic variable
- Static variable
- Call stack
- Dynamic memory allocation
- Stack buffer overflow
- Stack machine
- Stack overflow