x64 stack usage

Article • 08/03/2021

All memory beyond the current address of RSP is considered volatile: The OS, or a debugger, may overwrite this memory during a user debug session, or an interrupt handler. Thus, RSP must always be set before attempting to read or write values to a stack frame.

This section discusses the allocation of stack space for local variables and the **alloca** intrinsic.

Stack allocation

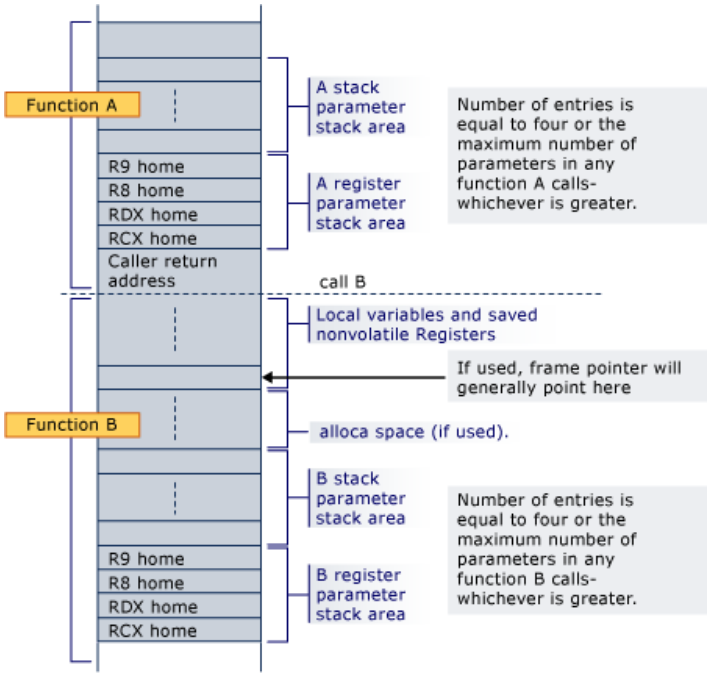
A function's prolog is responsible for allocating stack space for local variables, saved registers, stack parameters, and register parameters.

The parameter area is always at the bottom of the stack (even if `alloca` is used), so that it will always be adjacent to the return address during any function call. It contains at least four entries, but always enough space to hold all the parameters needed by any function that may be called. Note that space is always allocated for the register parameters, even if the parameters themselves are never homed to the stack; a callee is guaranteed that space has been allocated for all its parameters. Home addresses are required for the register arguments so a contiguous area is available in case the called function needs to take the address of the argument list (va_list) or an individual argument. This area also provides a convenient place to save register arguments during thunk execution and as a debugging option (for example, it makes the arguments easy to find during debugging if they are stored at their home addresses in the prolog code). Even if the called function has fewer than 4 parameters, these 4 stack locations are effectively owned by the called function, and may be used by the called function for other purposes besides saving parameter register values. Thus the caller may not save information in this region of stack across a function call.

If space is dynamically allocated (`alloca`) in a function, then a nonvolatile register must be used as a frame pointer to mark the base of the fixed part of the stack and that register must be saved and initialized in the prolog. Note that when `alloca` is used, calls to the same callee from the same caller may have different home addresses for their register parameters.

The stack will always be maintained 16-byte aligned, except within the prolog (for example, after the return address is pushed), and except where indicated in Function Types for a certain class of frame functions.

The following is an example of the stack layout where function A calls a non-leaf function B. Function A's prolog has already allocated space for all the register and stack parameters required by B at the bottom of the stack. The call pushes the return address and B's prolog allocates space for its local variables, nonvolatile registers, and the space needed for it to call functions. If B uses `alloca`, the space is allocated between the local variable/nonvolatile register save area and the parameter stack area.



When the function B calls another function, the return address is pushed just below the home address for RCX.

Dynamic parameter stack area construction

If a frame pointer is used, the option exists to dynamically create the parameter stack area. This is not currently done in the x64 compiler.

Function types

There are basically two types of functions. A function that requires a stack frame is called a *frame function*. A function that does not require a stack frame is called a *leaf function*.

A frame function is a function that allocates stack space, calls other functions, saves nonvolatile registers, or uses exception handling. It also requires a function table entry. A frame function requires a prolog and an epilog. A frame function can dynamically allocate stack space and can employ a frame pointer. A frame function has the full capabilities of this calling standard at its disposal.

If a frame function does not call another function then it is not required to align the stack (referenced in Section Stack Allocation).

A leaf function is one that does not require a function table entry. It can't make changes to any nonvolatile registers, including RSP, which means that it can't call any functions or allocate stack space. It is allowed to leave the stack unaligned while it executes.

malloc alignment

malloc is guaranteed to return memory that's suitably aligned for storing any object that has a fundamental alignment and that could fit in the amount of memory that's allocated. A *fundamental alignment* is an alignment that's less than or equal to the largest alignment that's supported by the implementation without an alignment specification. (In Visual C++, this is the alignment that's required for a `double`, or 8 bytes. In code that targets 64-bit platforms, it's 16 bytes.) For example, a four-byte allocation would be aligned on a boundary that supports any four-byte or smaller object.

Visual C++ permits types that have *extended alignment*, which are also known as *over-aligned* types. For example, the SSE types __m128 and `__m256`, and types that are declared by using `__declspec(align( n ))` where `n` is greater than 8, have extended alignment. Memory alignment on a boundary that's suitable for an object that requires extended alignment is not guaranteed by `malloc`. To allocate memory for over-aligned types, use _aligned_malloc and related functions.

alloca

_alloca is required to be 16-byte aligned and additionally required to use a frame pointer.

The stack that is allocated needs to include space after it for parameters of subsequently called functions, as discussed in Stack Allocation.

See also

x64 software conventions
align
__declspec

Feedback