

Re: HLA Lib

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2007-03/msg00752.html>

- *From:* "randyhyde@xxxxxxxxxxxxxx" <randyhyde@xxxxxxxxxxxxxx>
 - *Date:* 19 Mar 2007 20:31:33 -0700
-

On Mar 19, 6:45 pm, "vid...@xxxxxxxxxx" <vid...@xxxxxxxxxx> wrote:

Aw, that's a myth. All modern OSes guarantee that they clean this sort of stuff up. You don't want to get in the *habit* of not freeing up resources and so on, but when done *on purpose*, as is the case here, there's nothing wrong with it. The OS behavior is well-documented.

Where exactly is unmapping virtual memory guaranteed?

All memory allocation is freed up when the process quits. That's one of the major points of using protected address spaces in the first place. Otherwise, every program out there with a memory leak would bring down the OS after a while.

I think 64 bytes is too much. 16 bytes is probably about right (so that you're guaranteed 16-byte alignment for certain SSE operands). OTOH, HLA's memory manager actually does align to 16 bytes + 8, so getting 16-byte alignment is pretty easy -- just allocate 8 extra bytes and bump the returned address up by 8.

Still i think that 32bytes wouldn't cost you lot of memory, and will reduce need to resize blocks for 98% string operations.

I would agree that having 32-byte blocks would probably reduce the need to reallocate the block for around 98% of the string operations, but until I see modern research on the types of memory allocations that take place in HLA programs, it's dangerous to waste memory like that. It's a real shame to watch a programmer do things like use a 1-byte SCASD instruction to add 4 to EDI (to save a byte or two) and

Re: HLA Lib

then have ten times that amount wasted by a memory allocation call. Personally, I'm a bit concerned about the nine bytes of overhead that HLA strings already consume.

Then again, for string allocation it would be fairly easy to extend the space without affecting allocations for other objects. The HLA `stdlib` provides the `str.alloc` and `str.free` functions that are built on top of `mem.alloc` and `mem.free`. It would be a trivial exercise to modify `str.alloc` to allocate a little extra space for the string and then have `str.realloc` check to see if it really needs to do anything when asked to make the string larger.

OTOH, I'm sure you've read some of the complaints around here about "HLA does all this stuff behind your back." Having a `malloc` routine that allocates more storage than you really ask for, in anticipation of the user wanting to expand that block, really smacks of "doing stuff behind your back." If the user really expects to be resizing strings down the road, they can always explicitly ask for a larger block, or, perhaps, write a macro that wraps the `str.alloc` (or `mem.alloc`) function to reserve the extra space. Forcing all allocations to consume extra space is going to make the library less desirable to most people (and I'm not counting the usual crowd around here that loudly complains -- after all, they've made it clear they're not going to use library code in any way, shape, or form).

With 16bytes
you are probably going to hit few more.

That would need some testing.

Agreed.

Personally, I'd think that a `"mem.granularity(size)"` function would be the right way to go. Let the user set the granularity (to make it easy, require that size be a power of two). Then they could set it to whatever they want.

The problem with a large granularity on the allocation is that typical programs make a *lot* of small memory requests. In particular, the average string length you'll find in a program is quite small.

This is true for C++ and higher languages, and some C apps, but IMO not for typical assembly programs.

Re: HLA Lib

It is not typical for assembly language programs simply because so few assembly language programs have access to a decent heap manager. Most assembly language programs wind up calling something like VirtualAlloc or they reserve a ton of space in the BSS and carve that up. If those assembly language programmers had access to a decent heap manager, I'm sure they'd be using it rather than kludging up their own stuff. You do not, for example, see many HLA programs where the programmer micromanages the memory allocations. A few do, but not many.

Another example is an assembler. Consider FASM, for example. It makes *lots* of small memory requests.

It would be very ugly not to use custom allocation for array of static-sized objects, if memory size matters. :)

Then again, memory size really doesn't matter any more. So maybe all my debating is pointless. :-)

BTW, reallocations are one area where the HLA memory manager *isn't* all that great. To reduce fragmentation, the HLA memory manager allocates blocks from the end of the heap on down. This makes it unlikely that reallocation will simply consist of extending the size of the current block. OTOH, it *does* make the first-fit algorithm that HLA uses run incredibly fast because it almost always finds a block large enough just below the active end of the heap.

can't you just search from last block to first? that would have both advantages, wouldn't it?

No. The point of the filling in from the back is that in the average case, no search takes place. The largest block is usually the first block, and that almost always satisfies the memory request. Very, very fast (at least, until you reach the bottom of the heap, which most programs don't).

Re: HLA Lib

Actually, I *do* mention it in the documentation. I've just not gotten around to *posting* that documentation yet (that will come out with HLA stdlib v2.1, which I'll get back to work on after I get around to finishing HLA v1.90).

Still i dislike that kind of behavior. Reminds me of oldschool unix guys who need EOL at end of last line :))

Except this automatically puts that EOL there. :-)

I'm not sure I see the problem here. Unlike fileio, the stdin package generally strips and ignores the end-of-line sequence. This is an artifact of the buffering that stdin uses (versus the lack of buffering in fileio). If a line does not contain an end-of-line sequence, but butts right up against EOF, then the ReadLn function will supply a phantom EOLN and the stdin functions will process that last line properly. On the very next input operation, you *will* get an EOF exception raised. The user's program never sees that phantom newline; it exists solely for the correct operation of ReadLn.

Problem is if that function is used by anything else than readln. Especially by some user function. For example in some "wc -l" type of application...

Well, then we could debate whether that last line counts as a "line" even though it doesn't have a newline character at the end. I would argue "yes", you might argue "no". It's really just a matter of perspective.

In the end, keep in mind that a newline character (sequence) never appears in the stdin buffer. It's just a length-specified/zero-terminated string. The fact that ReadLn fakes a newline at EOF for it's own purposes doesn't change the semantics of the system whatsoever at all. That phantom newline *never* appears in the input stream that a user program would see. Indeed, all that really happens in the code is that a branch is taken. No data is copied around in any of the buffers; the linefeed exists only in a register so that a cmp and branch will happen appropriately down the road. I could have just as easily (though less efficiently) set a boolean flag and tested that.

The real question is "should EOF terminate some data input (such as an integer), or should it fail the process?" I've elected to have it

Re: HLA Lib

Re: HLA Lib

succeed (i.e., not do the old UNIX trick of requiring an EOLN at the end of the file). No extra data is ever read by the program.

Cheers,

Randy Hyde