Ling Internals

February 16, 2012

1 Heap structures

The heap represented by `heap_t` struct is a self-contained entity that can be used outside of a process context. Heap allocations and garbage collection does not depend on whether the heap is a process heap or not.

Upon initalization the heap receives references to the begining and the end of a static buffer. Allocations start in the initial buffer and, when it overflows, are spilled out to memory nodes.

To simplify a number of operations of the heap the initial buffer is represented as a fake memory node – `init_node`. The `init_node` is always the last element of the chain of memory nodes of the heap. The node is distinct from the node allocated using nalloc() in two aspects: (a) its threshold can not be obtained using NODE_THRESHOLD() macro and (b) it can not be freed using nfree(). The garbage collection checks whether it is dealing with `init_node` before attempting (a) or (b).

2 Garbage collection

The heap of the process is a list of memory nodes represented by `memnode_t` structures. The head of the list is pointed to by `heap->nodes`. The list is unidirectional linked using `next` field.

A single pass of GC operates on a single memory node. The last memory node that undergone GC is referenced by `proc->gc_spot` field. The idea is to run GC on all memory nodes one by one and start over when the end of the list is reached.

The root set consists of live registers and the stack. The preparation of registers for GC includes copying `r0` to `rs[0]`. The heap receives references to the registers and the stack as two generic term arrays: `rt_starts1/rt_ends1` and `rt_starts2\rt_ends2`.

Continuation pointers are intermixed with terms on stack. These pointers are tagged with 0x3 to make them look like immediate terms and thus be disregarded by GC.

The main GC routine is heap_collect_garbage() is called indirectly through heap_ensure() routine that combines possibly multiple runs of GC with memory allocation.

1. Determine the GC node. It should the next node after `gc_spot` or the first node of the chain of nodes.
2. Determine the size of the new memory node the data from the current node will be copied to. The size is the same as the size of GC node.
3. Check whether there is enough free space in the subsequent or the preceding node. If there is, then use that space. Otherwise, allocate a new memory node.
4. Determine the list of active nodes that can have references into the GC node. They may be sorted to provide for faster checks if the address belongs to one of the active nodes. Active nodes are just memory nodes that precede the GC node on the node chain.
5. Skip immediate terms. For the pointer terms determine where they are pointing to the GC node, to an active node, or somewhere else.
6. If the pointer term refers to the GC node, then check if the term is a "grave" and use the buried term if it is. If the term is not a grave, then copy the data to the new node according to its type tagging. Recursively invoke GC for any internal terms. Create a grave burying the copied term.
7. If the pointer term refers to an active node, then continue the process by examining any internal terms that may eventually lead to data in GC node.
8. If the pointer term refers to some other place, then keep it intact. It means that this is an older term that can not refer to GC node or it is a term from a literal pool.
9. If GC node is not `init_node` then remove the GC node from the list and destoy it. Update `heap->total_size` accordingly.
10. Set `heap->gc_spot` to the newly created memory node of the node the data has been copied to.

The variations of algorithm include the following: whether to sort active nodes or not, whether to merge with preceding/subsequent or both, how to select the GC node.

3 Bit operations

The bit manipulation in binaries uses a library of functions found in newbits.c. The two central structures of the library are `bit_sink_t` and `bit_src_t`. Both represents continuous bit arrays specialized for writing (`bit_sink_t`) and reading (`bit_src_t`).

`bit_sink_t` stores a pointer to the beginning and the length of the underlying octet array. The first and/or the last octet may be partially used. Such situations are captured by two counters: **nb** and **yb**. nb is the number of unused bits that should be skipped in the beginning of the first octet. yb gives the number of used bits in the last octet.

TODO