SBCL Internals

Table of Contents

1 Build

1.1 Cold init

At the start of cold init, the only thing the system can do is call functions, because `COLD-FSET` has arranged for that (and nothing else) to happen.

The tricky bit of cold init is getting the system to the point that it can run top level forms. To do that, we need to set up basic structures like the things you look symbols up in the structures which make the type system work.

So cold-init is the real bootstrap moment. Genesis dumps symbol<->package relationships but not the packages themselves, for instance. So we need to be able to make packages to fixup the system, but to do that we need to be able to make hash-tables, and to do that we need `RANDOM` to work, so we need to initialize the random-state and so on.

We could do much of this at genesis time, but it would just end up being fragile in a different way (needing to know about memory layouts of non-fundamental objects like packages, etc).

2 Calling Convention

The calling convention used within Lisp code on SBCL/x86 was, for the longest time, really bad. If it weren't for the fact that it predates modern x86 CPUs, one might almost believe it to have been designed explicitly to defeat the branch-prediction hardware therein. This chapter is somewhat of a brain-dump of information that might be useful when attempting to improve the situation further, mostly written immediately after having made a dent in the problem.

Assumptions about the calling convention are embedded throughout the system. The runtime knows how to call in to Lisp and receive a value from Lisp, the assembly-routines have intimate knowledge of what registers are involved in a call situation, '`src/compiler/target/call.lisp`' contains the VOPs involved in implementing function call/return, and '`src/compiler/ir2tran.lisp`' has assumptions about frame allocation and argument/return-value passing locations.

Note that most of this documentation also applies to other CPUs, modulo the actual registers involved, the displacement used in the single-value return convention, and the fact that they use the "old" convention anywhere it is mentioned.

2.1 Assembly Routines

```
;;; The :full-call assembly-routines must use the same full-call
;;; unknown-values return convention as a normal call, as some
;;; of the routines will tail-chain to a static-function. The
;;; routines themselves, however, take all of their arguments
;;; in registers (this will typically be one or two arguments,
;;; and is one of the lower bounds on the number of argument-
;;; passing registers), and thus don't need a call frame, which
;;; simplifies things for the normal call/return case. When it
;;; is neccessary for one of the assembly-functions to call a
;;; static-function it will construct the required call frame.
;;; Also, none of the assembly-routines return other than one
;;; value, which again simplifies the return path.
;;;    -- AB, 2006/Feb/05.
```

There are a couple of assembly-routines that implement parts of the process of returning or tail-calling with a variable number of values. These are `return-multiple` and `tail-call-variable` in '`src/assembly/x86/assem-rtns.lisp`'. They have their own calling convention for invocation from a VOP, but implement various block-move operations on the stack contents followed by a return or tail-call operation.

That's about all I have to say about the assembly-routines.

2.2 Local Calls

Calls within a block, whatever a block is, can use a local calling convention in which the compiler knows where all of the values are to be stored, and thus can elide the check for number of return values, stack-pointer restoration, etc. Alternately, they can use the full unknown-values return convention while trying to short-circuit the call convention. There is probably some low-hanging fruit here in terms of CPU branch-prediction.

The local (known-values) calling convention is implemented by the `known-call-local` and `known-return` VOPs.

Local unknown-values calls are handled at the call site by the `call-local` and `mutiple-call-local` VOPs. The main difference between the full call and local call protocols here is that local calls use a different frame setup protocol, and will tend to not use the normal frame layout for the old frame-pointer and return-address.

2.3 Full Calls

```
;;; There is something of a cross-product effect with full calls.
;;; Different versions are used depending on whether we know the
;;; number of arguments or the name of the called function, and
;;; whether we want fixed values, unknown values, or a tail call.
;;;
;;; In full call, the arguments are passed creating a partial frame on
;;; the stack top and storing stack arguments into that frame. On
;;; entry to the callee, this partial frame is pointed to by FP.
```

Basically, we use caller-allocated frames, pass an fdefinition, function, or closure in `EAX`, argcount in `ECX`, and first three args in `EDX`, `EDI`, and `ESI`. `EBP` points to just past the start of the frame (the first frame slot is at `[EBP-4]`, not the traditional `[EBP]`, due in part to how the frame allocation works). The caller stores the link for the old frame at `[EBP-4]` and reserved space for a return address at `[EBP-8]`. `[EBP-12]` appears to be an empty slot that conveniently makes just enough space for the first three multiple return values (returned in the argument passing registers) to be written over the beginning of the frame by the receiver. The first stack argument is at `[EBP-16]`. The callee then reallocates the frame to include sufficient space for its local variables, after possibly converting any `&rest` arguments to a proper list.

The above scheme was changed in 1.0.27 on x86 and x86-64 by swapping the old frame pointer with the return address and making EBP point two words later:

On x86/x86-64 the stack now looks like this (stack grows downwards):

```
----------
RETURN PC
----------
OLD FP
---------- <- FP points here
EMPTY SLOT
----------
FIRST ARG
----------
```

just as if the function had been CALLed and upon entry executed the standard prologue: PUSH EBP; MOV EBP, ESP. On other architectures the stack looks like this (stack grows upwards):

```
----------
FIRST ARG
----------
EMPTY SLOT
```

```
----------
RETURN PC
----------
OLD FP
---------- <- FP points here
```

2.4 Unknown-Values Returns

The unknown-values return convention consists of two parts. The first part is that of returning a single value. The second is that of returning a different number of values. We also changed the convention in 0.9.10, so we should describe both the old and new versions. The three interesting VOPs here are `return-single`, `return`, and `return-multiple`.

For a single-value return, we load the return value in the first argument-passing register (`A0`, or `EDI`), reload the old frame pointer, burn the stack frame, and return. The old convention was to increment the return address by two before returning, typically via a `JMP`, which was guaranteed to screw up branch- prediction hardware. The new convention is to return with the carry flag clear.

For a multiple-value return, we pass the first three values in the argument-passing registers, and the remainder on the stack. `ECX` contains the total number of values as a fixnum, `EBX` points to where the callee frame was, `EBP` has been restored to point to the caller frame, and the first of the values on the stack (the fourth overall) is at `[EBP-16]`. The old convention was just to jump to the return address at this point. The newer one has us setting the carry flag first.

The code at the call site for accepting some number of unknown- values is fairly well boilerplated. If we are expecting zero or one values, then we need to reset the stack pointer if we are in a multiple-value return. In the old convention we just encoded a `MOV ESP, EBX` instruction, which neatly fit in the two byte gap that was skipped by a single-value return. In the new convention we have to explicitly check the carry flag with a conditional jump around the `MOV ESP, EBX` instruction. When expecting more than one value, we need to arrange to set up default values when a single-value return happens, so we encode a jump around a stub of code which fakes up the register use convention of a multiple-value return. Again, in the old convention this was a two-byte unconditionl jump, and in the new convention this is a conditional jump based on the carry flag.

2.5 IR2 Conversion

The actual selection of VOPs for implementing call/return for a given function is handled in ir2tran.lisp. Returns are handled by `ir2-convert-return`, calls are handled by `ir2-convert-local-call`, `ir2-convert-full-call`, and `ir2-convert-mv-call`, and function prologues are handled by `ir2-convert-bind` (which calls `init-xep-environment` for the case of an entry point for a full call).

2.6 Additional Notes

The low-hanging fruit is going to be changing every call and return to use `CALL` and `RETURN` instructions instead of `JMP` instructions which is partly done on x86oids: a trampoline is `CALL`ed and that `JMP`s to the target which is sufficient to negate (most of?) the penalty.

A more involved change would be to reduce the number of argument passing registers from three to two, which may be beneficial in terms of our quest to free up a GPR for use on Win32 boxes for a thread structure.

Another possible win could be to store multiple return-values somewhere other than the stack, such as a dedicated area of the thread structure. The main concern here in terms of clobbering would be to make sure that interrupts (and presumably the internal-error machinery) know to save the area and that the compiler knows that the area cannot be live across a function call. Actually implementing this would involve hacking the IR2 conversion, since as it stands now the same argument conventions are used for both call and return value storage (same TNs).

3 Discriminating Functions

The Common Lisp Object System specifies a great deal of run-time customizeability, such as class redefinition, generic function and method redefinition, addition and removal of methods and redefinitions of method combinations. The additional flexibility defined by the Metaobject Protocol, specifying the generic functions called to achieve the effects of CLOS operations (and allowing many of them to be overridden by the user) makes any form of optimization seem intractable. And yet such optimization is necessary to achieve reasonable performance: the MOP specifies that a slot access looks up the class of the object, and the slot definition from that class and the slot name, and then invokes a generic function specialized on those three arguments. This is clearly going to act against the user's intuition that a slot access given an instance should be relatively fast.

The optimizations performed cannot be done wholly at compile-time, however, thanks to all of these possibilities for run-time redefinition and extensibility. This section describes the optimizations performed in SBCL's CLOS implementation in computing and calling the effective method for generic functions.

3.1 The Initial Discriminating Function

The system method on `SB-MOP:COMPUTE-DISCRIMINATING-FUNCTION`, under most circumstances, returns a function which closes over a structure of type `SB-PCL::INITIAL`, and which calls `SB-PCL::INITIAL-DFUN`. This discriminating function is responsible for implementing the computation of the applicable methods, the effective method, and thence the result of the call to the generic function. In addition, it implements optimization of these steps, based on the arguments it has been called with since the discriminating function was installed and the methods of the generic function.

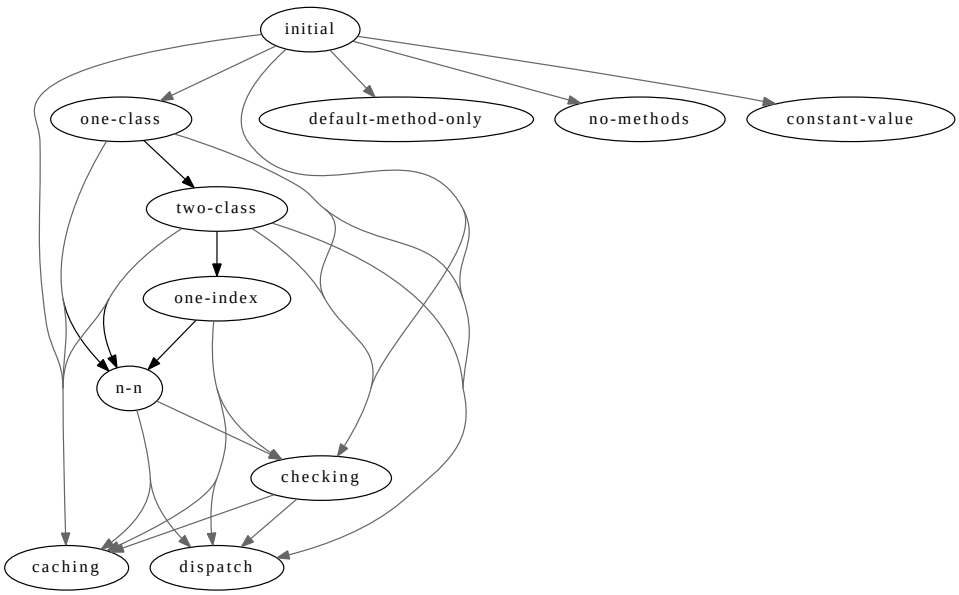

Figure 3.1

For each substantive change of the generic function (such as addition or removal of a method, or other reinitialization) the discriminating function is reset to its initial state.

The initial discriminating function can transition into a discriminating function optimized for the methods on the generic function (`SB-PCL::NO-METHODS`, `SB-PCL::DEFAULT-METHOD-ONLY`, `SB-PCL::CONSTANT-VALUE`), for slot access (`SB-PCL::ONE-CLASS`, `SB-PCL::TWO-CLASS`, `SB-PCL::ONE-INDEX`, `SB-PCL::N-N`[1]), or for dispatch based on its arguments (`SB-PCL::CACHING`, `SB-PCL::DISPATCH`). Those in the second category can transition into the third, or into a `SB-PCL::CHECKING` state where the choice between `SB-PCL::CACHING` and `SB-PCL::DISPATCH` has not yet been made.

The possible transitions are shown in Figure 3.1.

3.2 Method-Based Discriminating Functions

The method-based discriminating functions are used if all the methods of the generic function at the time of the first call are suitable: therefore, these discriminating function strategies do not transition into any of the other states unless the generic function is reinitialized. Of these discriminating functions, the simplest is the `SB-PCL::NO-METHODS`, which is appropriate when the generic function has no methods. In this case, the discriminating function simply performs an argument count check[2] and then calls `NO-APPLICABLE-METHOD` with the appropriate arguments.

If all of the specializers in all methods of the generic function are the root of the class hierarchy, `t`, then no discrimination need be performed: all of the methods are applicable on every call[3]. In this case, the `SB-PCL::DEFAULT-METHOD-ONLY` discriminating function can call the effective method directly, as it will be the same for every generic function call.[4]

If all methods of the generic function are known by the system to be side-effect-free and return constants, and the generic function has standard-method-combination and no eql-specialized methods, then the `SB-PCL::CONSTANT-VALUE` discriminating function can simply cache the return values for given argument types. Though this may initially appear to have limited applicability, type predicates are usually of this form, as in Example 3.1[5].

```
(defgeneric foop (x))
(defmethod foop ((foo foo)) t)
(defmethod foop (object) nil)
```

Example 3.1

More details of the cacheing mechanism are given in Section 3.5 [The Cacheing Mechanism], page 8 below.

[1] Would be better named as `M-N`, as there is no requirement for the number of classes and number of indices to be the same.

[2] Actually, this bit isn't currently done. Oops.

[3] Hm, there might be another problem with argument count here.

[4] I wonder if we're invalidating this right if we define a method on compute-applicable-methods...

[5] There is vestigial code in SBCL for a currently unused specialization of `SB-PCL::CONSTANT-VALUE` for boolean values only.

3.3 Accessor Discriminating Functions

Accessor Discriminating Functions are used when the effective method of all calls is an access to a slot, either reading, writing or checking boundness[6]; for this path to apply, there must be no non-standard methods on `SB-MOP:SLOT-VALUE-USING-CLASS` and its siblings. The first state is `SB-PCL::ONE-CLASS`, entered when one class of instance has been accessed; the discriminating function here closes over the wrapper of the class and the slot index, and accesses the slot of the instance directly.

If a direct instance of another class is passed to the generic function for slot access, then another accessor discriminating function is created: if the index of the slot in the slots vector of each instance is the same, then a `SB-PCL::TWO-CLASS` function is created, closing over the two class wrappers and the index and performing the simple dispatch. If the slot indexes are not the same, then we go to the `SB-PCL::N-N` state.

For slot accesses for more than two classes with the same index, we move to the `SB-PCL::ONE-INDEX` state which maintains a cache of wrappers for which the slot index is the same. If at any point the slot index for an instance is not the same, the state moves to `SB-PCL::N-N`, which maintains a cache of wrappers and their associated indexes; if at any point an effective method which is not a simple slot access is encountered, then the discriminating function moves into the `SB-PCL::CHECKING`, `SB-PCL::CACHING` or `SB-PCL::DISPATCH` states.

3.4 Cacheing and Dispatch Functions

`SB-PCL::CACHING` functions simply cache effective methods as a function of argument wrappers, while `SB-PCL::DISPATCH` functions have code that computes the actual dispatch. `SB-PCL::CHECKING` functions have a cache, but on cache misses will recompute whether or not to generate a `SB-PCL::CACHING` or `SB-PCL::DISPATCH` function.

(FIXME: I'm actually not certain about the above paragraph. Read the code again and see if it makes any more sense.)

3.5 The Cacheing Mechanism

In general, the cacheing mechanism works as follows: each class has an associated wrapper, with some number of uniformly-distributed random hash values associated with it; each cache has an associated index into this pseudovector of random hash values. To look a value up from a cache from a single class, the hash corresponding to the cache's index is looked up and reduced to the size of the cache (by bitmasking, for cache sizes of a power of two); then the entry at that index is looked up and compared for indentity with the wrapper in question. If it matches, this is a hit; otherwise the cache is walked sequentially from this index, skipping the 0th entry. If the original index is reached, the cache does not contain the value sought[7].

To add an entry to a cache, much the same computation is executed. However, if there is a collision in hash values, before the cache is grown, an attempt is made to fill the cache using a different index into the wrappers' hash values.

---

[6] Although there is ordinarily no way for a user to define a boundp method, some automatically generated generic functions have them.

[7] Actually, there's some kind of scope for overflow.

Wrappers are invalidated for caches by setting all of their hash values to zero. (Additionally, they are invalidated by setting their `depthoid` to -1, to communicate to structure type testers, and their `invalid` to non-`nil`, communicating to `obsolete-instance-trap`.

The hash value for multiple dispatch is computed by summing all of the individual hash values from each wrapper (excluding arguments for which all methods have `t` specializers, for which no dispatch computation needs to be done), jumping to the cache miss case if any wrapper has a zero hash index.

(FIXME: As of sbcl-0.9.x.y, the generality of multiple hash values per wrapper was removed, as it appeared to do nothing in particular for performance in real-world situations.)

References (O for working BibTeX):

The CLOS standards proposal

Kiczales and Rodruigez

AMOP

4 Foreign Linkage

4.1 Linkage-table

Linkage-table allows saving cores with foreign code loaded, and is also utilized to allow references to as-of-yet unknown aliens. See Section 4.2 [Lazy Alien Resolution], page 12.

The SBCL implementation is somewhat simplified from the CMUCL one by Timothy Moore, but the basic idea and mechanism remain identical: instead of having addresses from `dlsym(3)` in the core, we have addresses to an mmapped memory area (`LINKAGE_TABLE_SPACE`) that is initialized at startup to contain jumps & references to the correct addresses, based on information stored on the lisp side in `*LINKAGE-INFO*`.

4.1.1 Differences to CMUCL

CMUCL does lazy linkage for code, keeps all foreign addresses in the linkage-table, and handles the initialization from C. We do eager linkage for everything, maintain a separate `*STATIC-FOREIGN-SYMBOLS*` just like on non-linkage-table ports (this allows more code sharing between ports, makes thread-safety easier to achieve, and cuts one jump's worth of overhead from stuff like closure_tramp), and do the initialization from lisp.

4.1.2 Nitty Gritty Details

Symbols in `*STATIC-FOREIGN-SYMBOLS*` are handled the old fashioned way: linkage-table is only used for symbols resolved with `dlsym(3)`.

On system startup `FOREIGN-REINIT` iterates through the `*LINKAGE-INFO*`, which is a hash-table mapping dynamic foreign names to `LINKAGE-INFO` structures, and calls `arch_write_linkage_table_jmp/ref` to write the appropriate entries to the linkage-table.

When a foreign symbol is referred to, it is first looked for in the `*STATIC-FOREIGN-SYMBOLS*`. If not found, `ENSURE-FOREIGN-LINKAGE` is called, which looks for the corresponding entry in `*LINKAGE-INFO*`, creating one and writing the appropriate entry in the linkage table if necessary.

`FOREIGN-SYMBOL-ADDRESS` and `FOREIGN-SYMBOL-SAP` take an optional datap argument, used to indicate that the symbol refers to a variable. In similar fashion there is a new kind of fixup and a new VOP: `:FOREIGN-DATAREF` and `FOREIGN-SYMBOL-DATAREF-SAP`.

The `DATAP` argument is automagically provided by the alien interface for normal definitions, but is really needed only for dynamic foreign variables. For those it indicates the need for the indirection either within a conditional branch in `FOREIGN-SYMBOL-SAP`, or via `:FOREIGN-DATAREF` fixup and `FOREIGN-SYMBOL-DATAREF-SAP` VOP: "this address holds the address of the foreign variable, not the variable itself". Within SBCL itself (in the fixups manifest in various VOPs) this fixup type is never used, as all foreign symbols used internally are static.

One thing worth noting is that `FOREIGN-SYMBOL-SAP` and friends now have the potential side-effect of entering information in `*LINKAGE-INFO*` and the linkage-table proper. If the usage case is about checking if the symbol is available use `FIND-FOREIGN-SYMBOL-ADDRESS`, which is side-effect free. (This is used by SB-POSIX.)

4.1.3 Porting

4.1.3.1 Porting to new operating systems

Find a memory area for the linkage-table, and add it for the OS in 'src/compiler/target/parms.lisp' by defining SB!VM:LINKAGE-TABLE-SPACE-START and SB!VM:LINKAGE-TABLE-SPACE-END.
See existing ports and CMUCL for examples.

4.1.3.2 Porting to new architectures

Write arch_write_linkage_table_jmp and arch_write_linkage_table_ref.

Write FOREIGN-SYMBOL-DATAREF VOP.

Define correct SB!VM:LINKAGE-TABLE-ENTRY-SIZE in 'src/compiler/target/parms.lisp'.

4.2 Lazy Alien Resolution

On linkage-table ports SBCL is able to deal with forward-references to aliens – which is to say, compile and load code referring to aliens before the shared object containing the alien in question has been loaded.

This is handled by `ENSURE-DYNAMIC-FOREIGN-SYMBOL-ADDRESS`, which first tries to resolve the address in the loaded shared objects, but failing that records the alien as undefined and returns the address of a read/write/execute protected guard page for variables, and address of `undefined_alien_function` for routines. These are in turn responsible for catching attempts to access the undefined alien, and signalling the appropriate error.

These placeholder addresses get recorded in the linkage-table.

When new shared objects are loaded `UPDATE-LINKAGE-TABLE` is called, which in turn attempts to resolve all currently undefined aliens, and registers the correct addresses for them in the linkage-table.

4.3 Callbacks

SBCL is capable of providing C with linkage to Lisp – the upshot of which is that C-functions can call Lisp functions thru what look like function pointers to C.

These "function pointers" are called Alien Callbacks. An alien callback sequence has 4 parts / stages / bounces:

- Assembler Wrapper

  saves the arguments from the C-call according to the alien-fun-type of the callback, and calls #'ENTER-ALIEN-CALLBACK with the index indentifying the callback, a pointer to the arguments copied on the stack and a pointer to return value storage. When control returns to the wrapper it returns the value to C. There is one assembler wrapper per callback.[1] The SAP to the wrapper code vector is what is passed to foreign code as a callback.

  The Assembler Wrapper is generated by `ALIEN-CALLBACK-ASSEMBLER-WRAPPER`.

- #'ENTER-ALIEN-CALLBACK

  pulls the Lisp Trampoline for the given index, and calls it with the argument and result pointers.

- Lisp Trampoline

  calls the Lisp Wrapper with the argument and result pointers, and the function designator for the callback. There is one lisp trampoline per callback.

- Lisp Wrapper

  parses the arguments from stack, calls the actual callback with the arguments, and saves the return value at the result pointer. The lisp wrapper is shared between all the callbacks having the same same alien-fun-type.

[1] As assembler wrappers need to be allocated in static addresses and are (in the current scheme of things) never released it might be worth it to split it into two parts: per-callback trampoline that pushes the index of the lisp trampoline on the stack, and jumps to the appropriate assembler wrapper. The assembler wrapper could then be shared between all the callbacks with the same alien-fun-type. This would amortize most of the static allocation costs between multiple callbacks.

5 Funcallable Instances

5.1 Overview of Funcallable Instances

Funcallable instances in SBCL are implemented as a subtype of `function`, and as such must be directly funcallable using the same calling sequence as ordinary functions and closure objects, which means reading the first word of the object after the header, and then jumping to it (with an offset on non-x86 platforms). It must be possible to set the function of a funcallable instance, as CLOS (one user of funcallable instances) computes and sets the discriminating function for generic functions with `sb-mop:set-funcallable-instance-function`, and also allows the user to do the same.

Additionally, although this functionality is not exported to the normal user, they must support an arbitrary number of slots definable with `!defstruct-with-alternate-metaclass`. If generic functions were the only users of funcallable instances, then this might be less critical, but (as of SBCL 0.9.17) other users of funcallable instances are: the `ctor` make-instance optimization; the `method-function` funcallable instance which does the bookkeeping for fast method function optimization; and interpreted functions in the full evaluator.

5.2 Implementation of Funcallable Instances

The first word after the header of a funcallable instance points to a dedicated trampoline function (known as `funcallable_instance_tramp` in SBCL 0.9.17) which is responsible for calling the funcallable instance function, kept in the second word after the header. The remaining words of a funcallable instance are firstly the `layout`, and then the slots.

The implementation of funcallable instances inherited from CMUCL differed in that there were two slots for the function: one for the underlying `simple-fun`, and one for the function itself (which is distinct from the `simple-fun` in the case of a closure. This, coupled with an instruction in the prologue of a closure's function to fetch the function from the latter slot, allowed a trampolineless calling sequence for funcallable instances; however, drawbacks included the loss of object identity for the funcallable instance function (if a funcallable instance was set as the function of another, updates to the first would not be reflected in calls to the second) and, more importantly, a race condition in calling funcallable instances from one thread while setting its funcallable instance function in another. The current implementation, described in the paragraph above, does not suffer from these problems (the function of a funcallable instance can be set atomically and retains its identity) at the cost of an additional layer of indirection.

6 Objects In Memory

6.1 Type tags

The in-memory representation of Lisp data includes type information about each object. This type information takes the form of a lowtag in the low bits of each pointer to heap space, a widetag for each boxed immediate value and a header (also with a widetag) at the start of the allocated space for each object. These tags are used to inform both the GC and Lisp code about the type and allocated size of Lisp objects.

6.1.1 Lowtags

Objects allocated on the Lisp heap are aligned to a double-word boundary, leaving the low-order bits (which would normally identify a particular octet within the first two words) available for use to hold type information. This turns out to be three bits on 32-bit systems and four bits on 64-bit systems.

Of these 8 or 16 tags, we have some constraints for allocation:

- We need 6 of the low 8 bits of the word for widetags, meaning that one out of every four lowtags must be an `other-immediate` lowtag.
- We have four pointer types. Instance (struct and CLOS) pointers, function pointers, list pointers, and other pointers.
- `fixnum`s are required to have their lowtags be comprised entirely of zeros.
- There are additional constraints around the ordering of the pointer types, particularly with respect to list pointers (the NIL-cons hack).

Complicating this issue is that while the lowtag *space* is three or four bits wide, some of the lowtags are effectively narrower. The `other-immediate` tags effectively have a two-bit lowtag, and `fixnum`s have historically been one bit narrower than the other lowtags (thus `even-fixnum-lowtag` and `odd-fixnum-lowtag`) though with the recent work on wider fixnums on 64-bit systems this is no longer necessarily so.

The lowtags are specified in '`src/compiler/generic/early-objdef.lisp`'.

6.1.1.1 Fixnums

`Fixnum`s are signed integers represented as immediate values. In SBCL, these integers are `(- n-word-bits n-fixnum-tag-bits)` bits wide, stored in the most-significant section of a machine word.

The reason that `fixnum` tags are required to have the low `n-fixnum-tag-bits` as zeros is that it allows for addition and subtraction to be performed using native machine instructions directly, and multiplication and division can be performed likewise using a simple shift instruction to compensate for the effect of the tag.

6.1.1.2 Other-immediates

`Other-immediate`s are the lowtag part of widetag values. Due to the constraints of widetag allocation, one out of every four lowtags must be a widetag (alternately, the width of the `other-immediate` lowtag is two bits).

6.1.1.3 Pointers

There are four different pointer lowtags, largely for optimization purposes.

- We have a distinct list pointer tag so that we can do a listp test by simply checking the pointer tag instead of needing to retrieve a header word for each `cons` cell. This effectively halves the memory cost of `cons` cells.
- We have a distinct instance pointer tag so that we do not need to check a header word for each instance when doing a type check. This saves a memory access for retrieving the class of an instance.
- We have a distinct function pointer tag so that we do not need to check a header word to determine if a given pointer is directly funcallable (that is, if the pointer is to a closure, a simple-fun, or a funcallable-instance). This saves a memory access in the type test prior to `funcall` or `apply` of a function object.
- We have one last pointer tag for everything else. Obtaining further type information from these pointers requires fetching the header word and dispatching on the widetag.

6.1.2 Widetags

Widetags are used for three purposes. First, to provide type information for immediate (non-pointer) data such as characters. Second, to provide "marker" values for things such as unbound slots. Third, to provide type information for objects stored on the heap.

Because widetags are used for immediate data they must have a lowtag component. This ends up being the `other-immediate` lowtags. For various reasons it was deemed convenient for widetags to be no more than eight bits wide, and with 27 or more distinct array types (depending on build-time configuration), seven numeric types, markers, and non-numeric heap object headers there ends up being more than 32 widetags required (though less than 64). This combination of factors leads to the requirement that one out of every four lowtags be an `other-immediate` lowtag.

As widetags are involved in type tests for non-CLOS objects, their allocation is carefully arranged to allow for certain type tests to be cheaper than they might otherwise be.

- The numeric types are arranged to make `rational`, `float`, `real`, `complex` and `number` type tests become range tests on the widetag.
- The array types are arranged to make various type tests become range tests on the widetag.
- The string types have disjoint ranges, but have been arranged so that their ranges differ only by one bit, allowing the `stringp` type test to become a masking operation followed by a range test or a masking operation followed by a simple comparison.
- There may be other clevernesses, these are just what can be found through reading the comments above the widetag definition.

The widetags are specified in '`src/compiler/generic/early-objdef.lisp`'.

6.2 Heap Object Layout

Objects stored in the heap are of two kinds: those with headers, and cons cells. If the first word of an object has a header widetag then the object has the type and layout associated with that widetag. Otherwise, the object is assumed to be a `cons` cell.

Some objects have "unboxed" words without any associated type information as well as the more usual "boxed" words with lowtags. Obvious cases include the specialized array types, some of the numeric types, `system-area-pointer`s, and so on.

The primitive object layouts are specified in '`src/compiler/generic/objdef.lisp`'.

6.2.1 Header Values

As a widetag is only eight bits wide but a heap object header takes a full machine word, there is an extra 24 or 56 bits of space available for unboxed data storage in each heap object. This space is called the "header value", and is used for various purposes depending on the type of heap object in question.

6.2.2 Symbols

In contrast to the simple model of symbols provided in the Common Lisp standard, symbol objects in SBCL do not have a function cell. Instead, the mapping from symbols to functions is done via the compiler globaldb.

There are two additional slots associated with symbols. One is a hash value for the symbol (based on the symbol name), which avoids having to recompute the hash from the name every time it is required.

The other additional slot, on threaded systems only, is the TLS index, which is either `no-tls-value-marker-widetag` or an unboxed byte offset within the TLS area to the TLS slot associated with the symbol. Because the unboxed offset is aligned to a word boundary it appears as a `fixnum` when viewed as boxed data. It is not, in general, safe to increment this value as a `fixnum`, however, in case `n-fixnum-tag-bits` changes[1].

6.2.3 The NIL-cons Hack

As an "optimization", the symbol `nil` has `list-pointer-lowtag` rather than `other-pointer-lowtag`, and is aligned in memory so that the value and hash slots are the `car` and `cdr` of the `cons`, with both slots containing `nil`. This allows for `car` and `cdr` to simply do a lowtag test and slot access instead of having to explicitly test for `nil`, at the cost of requiring all symbol type tests and slot accesses to test for `nil`.

6.2.4 Functions and Code Components

All compiled code resides in `code-component` objects. These objects consist of a header, some number of boxed literal values, a "data block" containing machine code and `simple-fun` headers, and a "trace table" which is currently unused[2].

The `simple-fun` headers represent simple function objects (not `funcallable-instance`s or closures), and each `code-component` will typically have one for the main entry point and one per closure entry point (as the function underlying the closure, not the closure object proper). In a compiler trace-file, the `simple-fun` headers are all listed as entries in the IR2 component.

---

[1] This is not as unlikely as it might seem at first; while historically `n-fixnum-tag-bits` has always been the same as `word-shift` there is a branch where it is permitted to vary at build time from `word-shift` to as low as 1 on 64-bit ports, and a proposed scheme to allow the same on 32-bit ports

[2] Trace tables were originally used to support garbage collection using gengc in CMUCL. As there is still vestigial support for carrying them around at the end of `code-component`s, they may end up being used for something else in the future.

The `simple-fun` headers are held in a linked list per `code-component` in order to allow the garbage collector to find them during relocation. In order to be able to find the start of a `code-component` from a `simple-fun`, the header value is the offset in words from the start of the `code-component` to the start of the `simple-fun`.

7 Signal handling

7.1 Groups of signals

There are two distinct groups of signals.

7.1.1 Synchronous signals

This group consists of signals that are raised on illegal instruction, hitting a protected page, or on a trap. Examples from this group are: `SIGBUS/SIGSEGV`, `SIGTRAP`, `SIGILL` and `SIGEMT`. The exact meaning and function of these signals varies by platform and OS. Understandably, because these signals are raised in a controllable manner they are never blocked or deferred.

7.1.2 Asynchronous or blockable signals

The other group is of blockable signals. Typically, signal handlers block them to protect against being interrupted at all. For example `SIGHUP`, `SIGINT`, `SIGQUIT` belong to this group.

With the exception of `SIG_STOP_FOR_GC` all blockable signals are deferrable.

7.2 The deferral mechanism

7.2.1 Pseudo atomic sections

Some operations, such as allocation, consist of several steps and temporarily break for instance gc invariants. Interrupting said operations is therefore dangerous to one's health. Blocking the signals for each allocation is out of question as the overhead of the two `sigsetmask` system calls would be enormous. Instead, pseudo atomic sections are implemented with a simple flag.

When a deferrable signal is delivered to a thread within a pseudo atomic section the pseudo-atomic-interrupted flag is set, the signal and its context are stored, and all deferrable signals blocked. This is to guarantee that there is at most one pending handler in SBCL. While the signals are blocked, the responsibilty of keeping track of other pending signals lies with the OS.

On leaving the pseudo atomic section, the pending handler is run and the signals are unblocked.

7.2.2 `WITHOUT-INTERRUPTS`

Similar to pseudo atomic, `WITHOUT-INTERRUPTS` defers deferrable signals in its thread until the end of its body, provided it is not nested in another `WITHOUT-INTERRUPTS`.

Not so frequently used as pseudo atomic, `WITHOUT-INTERRUPTS` benefits less from the deferral mechanism.

7.2.3 Stop the world

Something of a special case, a signal that is blockable but not deferrable by `WITHOUT-INTERRUPTS` is `SIG_STOP_FOR_GC`. It is deferred by pseudo atomic and `WITHOUT-GCING`.

7.2.4 When are signals handled?

At once or as soon as the mechanism that deferred them allows.

First, if something is deferred by pseudo atomic then it is run at the end of pseudo atomic without exceptions. Even when both a GC request or a `SIG_STOP_FOR_GC` and a deferrable signal such as SIG_INTERRUPT_THREAD interrupts the pseudo atomic section.

Second, an interrupt deferred by WITHOUT-INTERRUPTS is run when the interrupts are enabled again. GC cannot interfere.

Third, if GC or `SIG_STOP_FOR_GC` is deferred by `WITHOUT-GCING` then the GC or stopping for GC will happen when GC is not inhibited anymore. Interrupts cannot delay a gc.

7.3 Implementation warts

7.3.1 Miscellaneous issues

Signal handlers automatically restore errno and fp state, but arrange_return_to_lisp_function does not restore errno.

7.3.2 POSIX – Letter and Spirit

POSIX restricts signal handlers to a use only a narrow subset of POSIX functions, and declares anything else to have undefined semantics.

Apparently the real reason is that a signal handler is potentially interrupting a POSIX call: so the signal safety requirement is really a re-entrancy requirement. We can work around the letter of the standard by arranging to handle the interrupt when the signal handler returns (see: `arrange_return_to_lisp_function`.) This does, however, in no way protect us from the real issue of re-entrancy: even though we would no longer be in a signal handler, we might still be in the middle of an interrupted POSIX call.

For some signals this appears to be a non-issue: `SIGSEGV` and other synchronous signals are raised by our code for our code, and so we can be sure that we are not interrupting a POSIX call with any of them.

For asynchronous signals like `SIGALARM` and `SIGINT` this is a real issue.

The right thing to do in multithreaded builds would probably be to use POSIX semaphores (which are signal safe) to inform a separate handler thread about such asynchronous events. In single-threaded builds there does not seem to be any other option aside from generally blocking asynch signals and listening for them every once and a while at safe points. Neither of these is implemented as of SBCL 1.0.4.

Currently all our handlers invoke unsafe functions without hesitation.

7.4 Programming with signal handling in mind

7.4.1 On reentrancy

Since they might be invoked in the middle of just about anything, signal handlers must invoke only reentrant functions or async signal safe functions to be more precise. Functions passed to `INTERRUPT-THREAD` have the same restrictions and considerations as signal handlers.

Destructive modification, and holding mutexes to protect desctructive modifications from interfering with each other are often the cause of non-reentrancy. Recursive locks are not likely to help, and while `WITHOUT-INTERRUPTS` is, it is considered untrendy to litter the code with it.

Some basic functionality, such as streams and the debugger are intended to be reentrant, but not much effort has been spent on verifying it.

7.4.2 More deadlocks

If functions A and B directly or indirectly lock mutexes M and N, they should do so in the same order to avoid deadlocks.

A less trivial scenario is where there is only one lock involved but it is acquired in a `WITHOUT-GCING` in thread A, and outside of `WITHOUT-GCING` in thread B. If thread A has entered `WITHOUT-GCING` but thread B has the lock when the gc hits, then A cannot leave `WITHOUT-GCING` because it is waiting for the lock the already suspended thread B has. From this scenario one can easily derive the rule: in a `WITHOUT-GCING` form (or pseudo atomic for that matter) never wait for another thread that's not in `WITHOUT-GCING`.

Somewhat of a special case, it is enforced by the runtime that `SIG_STOP_FOR_GC` and `SIG_RESUME_FROM_GC` always unblocked when we might trigger a gc (i.e. on alloc or calling into Lisp).

7.4.3 Calling user code

For the reasons above, calling user code, i.e. functions passed in, or in other words code that one cannot reason about, from non-reentrant code (holding locks), `WITHOUT-INTERRUPTS`, `WITHOUT-GCING` is dangerous and best avoided.

7.5 Debugging

It is not easy to debug signal problems. The best bet probably is to enable `QSHOW` and `QSHOW_SIGNALS` in runtime.h and once SBCL runs into problems attach gdb. A simple `thread apply all ba` is already tremendously useful. Another possibility is to send a SIGABORT to SBCL to provoke landing in LDB, if it's compiled with it and it has not yet done so on its own.

Note, that fprintf used by QSHOW is not reentrant and at least on x86 linux it is known to cause deadlocks, so place SHOW and co carefully, ideally to places where blockable signals are blocked. Use `QSHOW_SAFE` if you like.

8 Slot-Value

The ANSI Common Lisp standard specifies `slot-value`, `(setf slot-value)`, `slot-boundp` and `slot-makunbound` for standard-objects, and furthermore suggests that these be implemented in terms of Metaobject generic functions `slot-value-using-class`, `(setf slot-value-using-class)`, `slot-boundp-using-class` and `slot-makunbound-using-class`. To make performance of these operators tolerable, a number of optimizations are performed, at both compile-time and run-time[1].

8.1 Basic Implementation

All of the following, while described in terms of `slot-value`, also applies to `(setf slot-value)` and to `slot-boundp`, and could in principle be extended to `slot-makunbound`.

The basic implementation of `slot-value`, following the suggestion in the standards document, is shown in Example 8.1; the implementation of the other slot operators is similar. The work to be done simply to arrive at the generic function call is already substantial: we need to look up the object's class and iterate over the class' slots to find a slot of the right name, only then are we in a position to call the generic function which implements the slot access directly.

```
(defun slot-value (object slot-name)
  (let* ((class (class-of object))
         (slot-definition (find-slot-definition class slot-name)))
    (if (null slot-definition)
        (values (slot-missing class object slot-name 'slot-value))
        (slot-value-using-class class object slot-definition))))
```

Example 8.1

The basic implementation of `slot-value-using-class` specialized on the standard metaobject classes is shown in Example 8.2. First, we check for an obsolete instance (that is, one whose class has been redefined since the object was last accessed; if it has, the object must be updated by `update-instance-for-redefined-class`); then, we acquire the slot's storage location from the slot definition, the value from the instance's slot vector, and then after checking the value against the internal unbound marker, we return it.

[1] Note that ,at present, `slot-makunbound` and `slot-makunbound-using-class` are not optimized in any of the ways mentioned below.

```
(defmethod slot-value-using-class
    ((class std-class)
     (object standard-object)
     (slotd standard-effective-slot-definition))
  (check-obsolete-instance object)
  (let* ((location (slot-definition-location slotd))
         (value
          (etypecase location
            (fixnum (clos-slots-ref (instance-slots object) location))
            (cons (cdr location)))))
    (if (eq value +slot-unbound+)
        (values (slot-unbound class object (slot-definition-name slotd)))
        value)))
```

Example 8.2

Clearly, all of this activity will cause the performance of clos slot access to compare poorly with structure slot access; while there will be of necessity a slowdown between the slot accesses because the structure class need not be redefineable (while redefinition of standard-object classes is extremely common), the overhead presented in the above implementation is excessive.

8.2 Compiler Transformations

The compiler can assist in optimizing calls to `slot-value`: in particular, and despite the highly-dynamic nature of CLOS, compile-time knowledge of the name of the slot being accessed permits precomputation of much of the access (along with a branch to the slow path in case the parameters of the access change between compile-time and run-time).

8.2.1 Within Methods

If the object being accessed is a required parameter to the method, where the parameter variable is unmodified in the method body, and the slot name is a compile-time constant, then fast slot access can be supported through *permutation vectors*.

(FIXME: what about the metaclasses of the object? Does it have to be standard-class, or can it be funcallable-standard-class? Surely structure-class objects could be completely optimized if the class definition and slot name are both known at compile-time.)

Permutation vectors are built up and maintained to associate a compile-time index associated with a slot name with an index into the slot vector for a class of objects. The permutation vector applicable to a given method call (FIXME: or effective method? set of classes? something else?) is passed to the method body, and slots are accessed by looking up the index to the slot vector in the permutation vector, then looking up the value from the slot vector. (FIXME: a diagram would help, if I understood this bit well enough to draw a diagram).

Subsequent redefinitions of classes or of methods on `slot-value-using-class` cause an invalid index to be written into the permutation vector, and the call falls back to a full call to `slot-value`.

If the conditions for (structure or) permutation vector slot access optimization are not met, optimization of `slot-value` within methods falls back to the same as for calls to `slot-value` outside of methods, below.

8.2.2 Outside of Methods

A call to `slot-value` with a compile-time constant slot *name* argument is compiled into a call to a generic function named `(sb-pcl::slot-accessor :global` *name* `sb-pcl::reader)`, together with code providing load-time assurance (via `load-time-value`) that the generic function is bound and has a suitable accessor method. This generic function then benefits from the same optimizations as ordinary accessors, described in Section 3.3 [Accessor Discriminating Functions], page 8.

(FIXME: how does this get invalidated if we later add methods on `slot-value-using-class`? Hm, maybe it isn't. I think this is probably a bug, and that adding methods to `slot-value-using-class` needs to invalidate accessor caches. Bah, humbug. Test code in Example 8.3, and note that I think that the analogous case involving adding or removing methods from `compute-applicable-methods` is handled correctly by `update-all-c-a-m-gf-info`.)

```
(defclass foo () ((a :initform 0)))
(defun foo (x) (slot-value x 'a))
(foo (make-instance 'foo)) ; => 0
(defmethod slot-value-using-class :after
  ((class std-class) (object foo)
   (slotd standard-effective-slot-definition))
  (print "hi"))
(foo (make-instance 'foo)) ; => 0, no print
(defclass bar (foo) ((a :initform 1)))
(foo (make-instance 'bar)) ; => 1  and prints "hi"
(foo (make-instance 'foo)) ; => 0, no print
```

Example 8.3

8.3 MOP Optimizations

Even when nothing is known at compile-time about the call to `slot-value`, it is possible to do marginally better than in Example 8.2. Each effective slot definition metaobject can cache its own effective method, and the discriminating function for `slot-value-using-class` is set to simply call the function in its slot definition argument.

(FIXME: I'm pretty sure this is a bad plan in general. Or rather, it's probably a good plan, but the effective methods should probably be computed lazily rather than eagerly. The default image has 8589 closures implementing this optimization: 3 (`slot-value`, `set-slot-value` and `slot-boundp`) for each of 2863 effective slots.)

(Also note that this optimization depends on not being able to specialize the `new-value` argument to `(setf slot-value-using-class)`.)

9 Specials

9.1 Overview

Unithread SBCL uses a shallow binding scheme: the current value of a symbol is stored directly in its value slot. Accessing specials is pretty fast but it's still a lot slower than accessing lexicals.

With multithreading it's slightly more complicated. The symbol's value slot contains the global value and each symbol has a `TLS-INDEX` slot that - when it's first bound - is set to a unique index of the thread local area reserved for this purpose. The tls index is initially zero and at index zero in the tls `NO-TLS-VALUE-MARKER` resides. `NO-TLS-VALUE-MARKER` is different from `UNBOUND-MARKER` to allow `PROGV` to bind a special to no value locally in a thread.

9.2 Binding and unbinding

Binding goes like this: the binding stack pointer (bsp) is bumped, old value and symbol are stored at bsp - 1, new value is stored in symbol's value slot or the tls.

Unbinding: the symbol's value is restored from bsp - 1, value and symbol at bsp - 1 are set to zero, and finally bsp is decremented.

The `UNBIND-TO-HERE` VOP assists in unwinding the stack. It iterates over the bindings on the binding stack until it reaches the prescribed point. For each binding with a non-zero symbol it does an `UNBIND`.

How can a binding's symbol be zero? `BIND` is not pseudo atomic (for performance reasons) and it can be interrupted by a signal. If the signal hits after the bsp is incremented but before the values on the stack are set the symbol is zero because a thread starts with a zeroed tls plus `UNBIND` and `UNBIND-TO-HERE` both zero the binding being unbound.

Zeroing the binding's symbol would not be enough as the binding's value can be moved or garbage collected and if the above interrupt initiates gc (or be `SIG_STOP_FOR_GC`) it will be greeted by a garbage pointer.

Furthermore, `BIND` must always write the value to the binding stack first and the symbol second because the symbol being non-zero means validity to `UNBIND-TO-HERE`. For similar reasons `UNBIND` also zeroes the symbol first. But if it is interrupted by a signal that does an async unwind then `UNBIND-TO-HERE` can be triggered when the symbol is zeroed but the value is not. In this case `UNBIND-TO-HERE` must zero out the value to avoid leaving garbage around that may wreck the ship on the next `BIND`.

In other words, the invariant is that the binding stack above bsp only contains zeros. This makes `BIND` safe in face of gc triggered at any point during its execution.

10 Character and String Types

The :SB-UNICODE feature implies support for all 1114112 potential characters in the character space defined by the Unicode consortium, with the identity mapping between lisp char-code and Unicode code point. SBCL releases before version 0.8.17, and those without the :SB-UNICODE feature, support only 256 characters, with the identity mapping between char-code and Latin1 (or, equivalently, the first 256 Unicode) code point.

In the absence of the :SB-UNICODE feature, the types base-char and character are identical, and encompass the set of all 256 characters supported by the implementation. With the :SB-UNICODE on *features* (the default), however, base-char and character are distinct: character encompasses the set of all 1114112 characters, while base-char represents the set of the first 128 characters.

The effect of this on string types is that an sbcl configured with :SB-UNICODE has three disjoint string types: (vector nil), base-string and (vector character). In a build without :SB-UNICODE, there are two such disjoint types: (vector nil) and (vector character); base-string is identially equal to (vector character).

The SB-KERNEL:CHARACTER-SET-TYPE represents possibly noncontiguous sets of characters as lists of range pairs: for example, the type standard-char is represented as the type (sb-kernel:character-set '((10 . 10) (32 . 126)))

10.1 Memory Layout

Characters are immediate objects (that is, they require no heap allocation) in all permutations of build-time options. Even on a 32-bit platform with :SB-UNICODE, there are three bits to spare after allocating 8 bits for the character widetag and 21 for the character code. There is only one such layout, and consequently only one widetag is needed: the difference between base-char and character is purely on the magnitude of the char-code.

Objects of type (simple-array nil (*)) are represented in memory as two words: the first is the object header, with the appropriate widetag, and the second is the length field. No memory is needed for elements of these objects, as they can have none.

Objects of type simple-base-string have the header word with widetag, then a word for the length, and after that a sequence of 8-bit char-code bytes. The system arranges for there to be a null byte after the sequence of lisp character codes.

Objects of type (simple-array character (*)), where this is a distinct type from simple-base-string, have the header word with widetag, length, and then a sequence of 32-bit char-code bytes. Again, the system arranges for there to be a null word after the sequence of character codes.

Non-simple character arrays, and simple character arrays of non-unit dimensionality, have an array header with a reference to an underlying data array of the appropriate form from the above representations.

10.2 Reader and Printer

The " reader macro always constructs an object of type (simple-array character), even if all of the characters within the quotation marks are of type base-char. This implies

that only strings of type `(vector character)` will be able to be printed when `*print-readably*` is true: attempting to print strings of other types will cause an error of type `print-not-readable`.

11 Threads

11.1 Implementation (Linux x86/x86-64)

Threading is implemented using pthreads and some Linux specific bits like futexes.

On x86 the per-thread local bindings for special variables is achieved using the %fs segment register to point to a per-thread storage area. This may cause interesting results if you link to foreign code that expects threading or creates new threads, and the thread library in question uses %fs in an incompatible way. On x86-64 the r12 register has a similar role.

Queues require the `sys_futex` system call to be available: this is the reason for the NPTL requirement. We test at runtime that this system call exists.

Garbage collection is done with the existing Conservative Generational GC. Allocation is done in small (typically 8k) regions: each thread has its own region so this involves no stopping. However, when a region fills, a lock must be obtained while another is allocated, and when a collection is required, all processes are stopped. This is achieved by sending them signals, which may make for interesting behaviour if they are interrupted in system calls. The streams interface is believed to handle the required system call restarting correctly, but this may be a consideration when making other blocking calls e.g. from foreign library code.

Large amounts of the SBCL library have not been inspected for thread-safety. Some of the obviously unsafe areas have large locks around them, so compilation and fasl loading, for example, cannot be parallelized. Work is ongoing in this area.