1 Boxed data representation.

Lisp data which encodes its type is said to be *boxed.* Boxed data is either *immediate* (as is the case of fixnums, characters, and some other objects used in the runtime system) or a pointer into an *allocated object.*

Allocated objects are aligned on 8-byte boundaries. If the first word of an allocated object is an *immheader*, the object is an *ivector*; if the first word is a *nodeheader*, the object is a *gvector*; otherwise, the object is a cons cell. Gvectors and ivectors are collectively referred to as uvectors.

Put another way: all non-immediate data other than cons cells are uvectors. (NIL is actually a bizarre special case; ignore it for now.)

A header (immheader or nodeheader) can't appear anywhere where a boxed value can (other than as the first word of a uvector.) There are no lisp primitives or compiler operations that return headers as values or cause them to appear in boxed machine registers. A uvector's header is a 32-bit value whose upper 24 bits denote the uvector's length (in elements) and whose low 8 bits (called the *subtag*) denote the uvector's type. (Uvectors can therefore contain at most (2^24)-1 elements.)

The 3 least-significant bits of a boxed object are used to encode the object's *primary* data type. In some cases, it's necessary to examine all three of these bits (the object's *fulltag*); in many cases, it's sufficient to examine only the two least-significant bits (the object's *lisptag)* in order to determine the object's primary data type.

Fulltag values used in the ppccl implementation are as follows:

Fulltag	Constant Name	Comment
#bxxxxx...x000	ppc::fulltag-even-fixnum	Even fixnum
#bxxxxx...x001	ppc::fulltag-cons	A pointer to a cons cell
#bxxxxx...x010	ppc::fulltag-nodeheader	Used in gvector headers, see below
#bxxxxx...x011	ppc::fulltag-imm	Small scalar values, e.g, characters
#bxxxxx...x100	ppc::fulltag-odd-fixnum	Odd fixnum
#bxxxxx...x101	ppc::fulltag-nil	NIL
#bxxxxx...x110	ppc::fulltag-misc	A pointer to a uvector
#bxxxxx...x111	ppc::fulltag-immheader	Used in ivector headers, see below

And lisptag values are:

Lisptag	Constant Name	Comment
#bxxxxxx...x00	ppc::tag-fixnum	All fixnums
#bxxxxxx...x01	ppc::tag-list	A pointer to a cons cell or NIL
#bxxxxxx...x10	ppc::tag-misc	A pointer to a uvector
#bxxxxxx...x11	ppc::tag-imm	An immediate object

An object's *typecode* is the value of its lisptag if that value is other than ppc::tag-misc and the value of the uvector's header's subtag otherwise. Determining an object's typecode therefore takes about 5 PPC instructions:

```
;; set DST to a fixnum representing the typecode of SRC, using
;; TMP as an (unboxed) temporary:
    clrlslwi dst,src,30,2 ; clear left 30 bits of src, shift
                          ; store in dst
    cmpwi dst,'ppc::tag-misc
    bne 1f
    lbz tmp,ppc::misc-subtag-offset(src); load (unboxed) subtag
    slwi dst,tmp,2; box result
1:
```

2 Characteristics of primitive data types

Fixnums are effectively signed 30-bit integers. Addition and subtraction can be performed inline; multiplication requires that one operand be unboxed (arithmetically shifted to the right two bits.)

Code-vectors are ivectors which must contain valid PPC instructions (and UUOs, which are special trap instructions used by the runtime system.) When code-vectors are allocated in memory (as during incremental compilation, fasloading, initial image loading), the processor instruction and data caches must be flushed and synchronized to ensure that the instruction cache is loaded from memory. (The runtime system doesn't use self-modifying code, but architectures with separate I and D caches often impose this restriction on dynamically allocated code.) Code-vectors typically contain many calls to "subprimitives" (runtime support routines); these are typically implemented as absolute branch instructions to a target whose address varies from session to session.

Functions are gvectors whose first element is a code-vector and whose other elements are constants referenced by that code vector and metainformation about the function (its name, arguments count, and other characteristics.) For historical reasons, functions are sometimes called "lfuns" (for Lisp FUNctions, as if there were some other kind.)

Double-floats are ivectors whose (32-bit unsigned) word count is 3. The 64-bit double-float value is stored in the 2nd and 3rd words (to enforce doubleword alignment.)

Macptrs (should change the name) are objects which encode foreign addresses.

Bignums are arrays of signed 32-bit values; the least-significant word of the value is stored at the lowest address. The sign bit of the most significant word must match the sign of the integer: the most-negative 32-bit integer would therefore be represented as the one-word bignum whose only word was #x80000000; the unsigned value #x80000000 would be represented as the two-word bignum #x80000000 #x00000000.

In a bitvector, bit 0 is the most-significant bit of the 0th word.

There could be 32 ivector subtags and 32 gvector subtags, but some possible values are unused in order to make certain operations faster. Ivector and gvector

subtags are ordered so that:

- all numeric subtags are arithmetically lower than non-numeric subtags; there are further orderings such that integer < ratio < float < complex.
- all ivector subtags whose element size is 32 bits are arithmetically lower than those whose element size is other than 32 bits.
- all vector subtags are arithmetically greater than all non-vector subtags.
- all array subtags are arithmetically greater than all non-array subtags.

NIL is a sort of "misaligned cons cell" - it straddles two doublewords in memory. NIL is always allocated in a "static" area of memory; its address doesn't change within a given lisp session but may change from session to session. (In fact, MkLinux and LinuxPPC tend to allocate application memory differently, so the address of NIL varies even under Linux.) There are a few dozen symbols which the kernel needs to access that are allocated at known fixed offsets from NIL. (One such symbol is called "nilsym" : it's used to represent NIL as a symbol - its plist, pname, symbol-function, symbol-package, etc.) T is allocated exactly 17 bytes from NIL; this is sometimes exploited in code sequences that map some condition to a lisp boolean value without introducing conditional branches. Symbolp has to special-case NIL (an object is a symbol iff it's NIL or if its typecode is PPC::SUBTAG-SYMBOL.)

(Proper) symbols are 5-element gvectors (so, including the header they each take up 24 bytes.) Symbols which are not globally FBOUNDP contain objects that signal "undefined function" errors when called in their function cells. It is always safe to call a known symbol without an explicit FBOUNDP check (safe in the sense that it won't sail off into the ozone.)

3 PPC register usage

3.1 GC considerations

Garbage collection is always invoked via an exception (typically, the exception is caused by a write to a protected guard page, but even calling the GC explicitly via (CCL:GC) causes a trap instruction to be executed.) The garbage collector must examine the registers (saved by the OS exception mechanism) of each lisp thread and must be able to reliably know which registers contain tagged lisp objects and which do not. (To confuse matters slightly, 2 GPRs and 2 SPRs are allowed to contain "locatives" : word-aligned pointers into the middle of certain uvectors.)

The GC assumes that R16-R31 contain tagged lisp objects. These registers will be treated as roots by the GC and it may change the value of any such register if the corresponding object is relocated.

The "loc_g" register (R14) is treated as a tagged object if its tag is not TAG_FIXNUM; otherwise, it is treated as a pointer INTO a gvector. (Such pointers are used internally to note intergenerational stores.)

The "loc_pc" register (R15) and the CTR and LR are all treated as tagged objects if their tags are not TAG_FIXNUM and are treated as a pointer into a code-vector otherwise. Certain values in stack frames are also assumed to contain pc-locatives (return addresses in the call chain), and OS exception frames also contain the PC (SRR0 at the time of the exception) which must be treated as a pc-locative.

Usually, some other root will be pointing AT the object that a locative points INTO; the mark phase of the GC will usually encounter these other references before it encounters the locative. If the mark phase encounters a locative into an as-yet unmarked object, it scans backward from the locative until it finds the object's header. For gvector locatives, this is guaranteed to succeed because no gvector can contain an "embedded" gvector header. For pc locatives, it's guaranteed to succeed because no machine instruction can "look like" a code-vector header. (This restriction is enforced by ensuring that no code vector has more than (2^18)-1 instructions; this guarantees that 6 most significant bits of the code-vector header are 0, and no valid PPC instruction can have a major opcode of 0.)

All other GPRs are assumed to contain unboxed values, though some (stack and heap pointers) have significance to the GC.

Some implications of these conventions include:

- all GPRs that are assumed to contain tagged data must actually contain tagged data at any time that a GC could occur. It is potentially catastrophic if a tagged register contains garbage that looks superficially like a valid tagged pointer.
- unboxed registers should never contain pointers to heap-allocated lisp objects. (More accurately, one shouldn't assume that the object's address will stay the same if a GC occurs, but it's safe to assume that the object's tag will be preserved.)
- it is wisest to assume that a GC could occur any time that lisp code is running; this would simplify eventual integration with a preemptive scheduler.

3.2 Calling conventions and compiled code

The following GPRs have dedicated usage when lisp code is executing.

Reg	Name	Usage	Comment
r0	rzero	always contains 0	Handy to have 0 in a register
r1	sp	control stack pointer	Same stack that C uses
r2	rnil	always contains NIL	Some constants allocated near NIL
r3-r7	imm0-imm4	unboxed temp regs	
r8	nargs	Argument/value count	Always boxed (fixnum)
r9	freeptr	Heap free pointer	Always doubleword-aligned
r10	initptr	"slow" free pointer	Old value of free pointer
r11	memo	EGC memo buffer pointer	Stack: grows down (towards 0)
r12	tsp	Temp stack pointer	Always doubleword-aligned
r13	vsp	Value stack pointer	Always word-aligned
r14	loc-g	Gvector locative	See above
r15	loc-pc	Codevector locative	See above
r16	fn	Current function	Function object
r17	temp3	Boxed temp reg	AKA "fname"
r18	temp2	Boxed temp reg	AKA "nfn"
r19	temp1	Boxed temp reg	AKA "next-method-context"
r20	temp0	Boxed temp reg	AKA "closure-data" (alias unused)
r21	arg_x	Boxed temp reg	Second-to-last arg durng call
r22	arg_y	Boxed temp reg	Next-to-last arg during call
r23	arg_z	Boxed temp reg	Last arg during call, return value
r24-r31	save7-save0	Boxed callee-save regs	save0=r31, save7=r24

Each lisp thread (sometimes called a "stack group") has 3 private stacks: a "control stack" used to save invocation history and to call foreign code, a "value stack" used to transmit some function arguments and return values and for temporary storage of other boxed lisp objects, and a "temp stack" used to allocate objects that have dynamic extent. All threads share common pointers to the heap and to the "EGC memoization buffer".

The value stack contains nothing but tagged lisp objects. (Actually, there's a case or two where it can contain certain types of gvector headers on doubleword boundaries.) Values are pushed on the value stack by executing:

```
stwu val,-4(vsp)
```

VPUSHing is atomic: at no point does the VSP point at stale or uninitialized data. (Actually, there are a few cases where it does, but not during a VPUSH.) Popping objects involves the 2-instruction sequence

```
lwz reg,0(vsp)
la vsp,4(vsp)
```

A sequence of VPOPs could defer or combine the "la" instructions.

Stack-overflow detection on the value stack is transparent, automatic, and incredibly hairy.

A set of GPRs can be saved on the vstack via:

```
la vsp,-n*4(vsp)     ; make room
stmw r(32-n),0(vsp) ; save regs
```

Note that if a GC were to take place between these instructions (because of a hypothetical preemptive scheduler), the GC would have to recognize that the n*4 bytes at the current value of the VSP contain uninitialized data.

Compiled lisp code doesn't use a frame pointer to access values on the vstack (everything is addressed relative to the VSP). The compiler's compile-time notion of what the top of the vstack looks like must therefore match reality or wackiness will ensue ...

The control stack contains a sequence of lisp frames. Each frame contains a backpointer to the previous (lisp or foreign) frame, the saved value of the "fn" register (the calling function object, in general), the return address, and the value that the VSP should be restored to on exit (used to discard incoming arguments.) A lisp frame can be distinguished from a foreign frame by virtue of the fact that it's always exactly 4 words long and the value of the calling "fn" register is saved where a raw (fixnum-tagged) return address would otherwise be saved.) There is again a "window" when a newly-allocated control stack frame is being initialized: the creation of a lisp control stack frame is not atomic with respect to a preemptive scheduler.

A global variable contains a limit for control-stack overflow detection. Any code that builds a new stack frame should check the value of SP against this limit.

```
lwz immtemp,csoverflow_limit(rnil)
twleu sp,immtemp
```

Only non-leaf functions need to build frames on the control stack. Leaf functions (and "near leaf functions") are very common in Lisp; a smarter compiler could improve performance quite a bit by being clever about this.

The temp stack contains a sequence of variable-length frames aligned on doubleword boundaries. The first word in each frame is a backpointer to the previous frame; the second word indicates whether the frame contains immediate or tagged data. The data itself starts on the next doubleword and extends to the previous frame. Initializing a frame on the tstack is not atomic with respect to a preemptive scheduler. Overflow detection on the tstack happens automatically if the frame size is less than 4k (the MMU page size.) Subprimitives should be used to allocate objects that might be larger (and to check for overflow while doing so.)

Calling a known function object involves:

1. Loading the function object into the nfn register (temp2, r18).
2. Loading the function's code vector into some other temporary node register

3. Moving the code vector into the "count" register (CTR)

4. Branching to the count register (possibly saving the return address in the LR.)

Note that the tag of the code vector is 6 (ppc::fulltag-misc); the BCTR[L] instruction ignores the low 2 bits of the CTR, so this has the effect of transferring control to the first instruction of the code vector (4 bytes past the code-vector header.)

Calling a globally named function involves loading the symbol into the "fname" node temporary, loading the function object from the symbol's function cell into the "nfn" register, and proceeding as above. If the symbol in question is not globally FBOUNDP, the code-vector will point to an instruction sequence that signals an error, using the value of the FNAME register to name the undefined function.

These temporary registers (nfn and fname) have defined meanings at the point of function call but can be used for any other purposes at other times.

To perform a function call (in the simplest, non-tail recursive case), the caller

1. If the function takes more than 3 arguments, vpushes the values of all but the last 3 arguments.

2. If the function takes 3 or more arguments, loads the next-to-next-to-last argument into the arg_x register.

3. If the function takes 2 or more arguments, loads the next-to-last argument into the arg_y register.

4. If the function takes at least one argument, loads the last argument into the arg_z register.

5. Sets the nargs register to the total number of outgoing arguments * 4 (i.e. as a fixnum)

6. Transfers control to the callee. Note that this always involves setting "nfn" to the called function object.

The callee:

1. Can use TWI instructions to check that the caller passed a valid argument count.

2. Can use the incoming value of "nargs" to determine default values for &optional ,&key &rest arguments.

3. Must save the incoming LR, FN, and SAVE[0-7] registers if it changes them,

4. Can use the incoming value of NFN to address constants. (Typically, the caller's value of FN is saved by the callee, which then copies NFN to FN and uses FN to address constants.)

5. In the single-value return case:
   (a) loads the return value into the arg_z register
   (b) restores any of LR, FN, SAVE[0-7] that may have been saved
   (c) restores the VSP to the value it had before the caller VPUSHed any outgoing arguments.
   (d) returns to the caller by branching to the LR ("blr").
6. In the multiple-value-return case
   (a) VPUSHes all return values and sets the nargs register to a (boxed) count of outgoing values
   (b) restores the callee-save registers
   (c) calls a subprim to return the multiple values if the caller requested them

There's additional complexity when the caller expects multiple values.

4 Heap Allocation

Newly created lisp objects are created in a block of memory called the dynamic heap. Conceptually, this block starts out empty; objects are allocated at the lowest address in this large free block and the "bottom" of the free space "moves up"past the last allocated object. The point at which new objects are created is called the freepointer; two registers (freeptr and initptr) point at this doubleword-aligned address whenever an object is to be allocated. Allocation generally involves:

1. rounding the object size (including header, if applicable) up to an even multiple of 8 bytes; call this the physical size of the object.
2. advancing the freeptr by the object's physical size
3. using the initptr to initialize the object, if necessary.
4. setting the initptr to the new value of the freeptr

If the dynamic heap was of infinite size, this would be all that we'd have to do. Since it isn't, it's necessary to check for heap overflow periodically. This is accomplished by write-protecting a set of 8 4Kbyte guard pages somewhere beyond the free pointer and advancing the freeptr via

```
stwu rzero,phys_size(freeptr)  ; physical size constant and <32K
stwux rzero,freeptr,physsizereg ; physical size <= 32K
```

(This mechanism works as long as the physical size is less than 32K bytes; in most applications, this is true of the overwhelming majority of cases.)

Memory between the free pointer and the guard page segment is zeroed; when a store to a heap guard page takes place, the fault handler can either

1. trigger a GC or
2. unprotect the current guard segment, zero its contents, protect the next segment, and resume execution. Zeroing large blocks of memorycan be accomplished efficiently via the use of DCBZ instructions (which zero an entire 32-byte cache block at time.)

Garbage collection will compact the live data in the allocated portion of the heap, typically (if any garbage was reclaimed) causing the free pointer to move back towards the "beginning" of the heap. On exit, the GC will set the freeptr and initptr to the new free pointer, zero the area between the freeptr and the next guard page segment, write-protect that segment, and resume execution at the faulting attempt to write to the new freeptr.

5 GC details

The GC uses a Mark/Compact algorithm; its execution time is essentially a factor of the amount of live data in the heap. (The somewhat better-known Mark/Sweep algorithms don't compact the live data but instead traverse the garbage to rebuild free-lists; their execution time is therefore a factor of the total heap size.)

The dynamic heap is allocated (vial a call to MALLOC or something similar) when the lisp starts up; its size is determined in some platform-specific way. Two related data structures are also allocated at this time:

1. the markbits bitvector, which contains a bit for every doubleword in the dynamic heap (plus a few extra words for alignment and so that sub-bitvectors can start on word boundaries.)
2. the relocation table, which contains a 32-bit word for every 32 doublewords in the dynamic heap, plus an extra word used to keep track of the end of the heap.

The total GC space overhead is therefore on the order of 3% (2/64 or 1/32).

The general algorithm proceeds as follows:

5.1 Mark phase.

Each doubleword in the dynamic heap has a corresponding bit in the markbits vector. (For any doubleword in the heap, the index of its mark bit is determined by subtracing the address of the start of the heap from the address of the object and dividing the result by 8.) The GC knows the markbit index of the

free pointer, so determining that the markbit index of a doubleword address is between the start of the heap and the free pointer can be done with a single unsigned comparison.

The markbits of all doublewords in the dynamic heap are zeroed before the mark phase begins. An object is *marked* if the markbits of all of its constituent doublewords are set and unmarked otherwise; setting an object's markbits involves setting the corrsponding markbits of all constituent doublewords in the object.

The mark phase traverses each root. If the fulltag of the value of the root is FULLTAG-MISC or FULLTAG-CONS and the object's doubleword address is within the dynamic heap, then:

1. If the object is already marked, do nothing.
2. Set the object's markbit(s).
3. If the object is an ivector, do nothing further.
4. If the object is a cons cell, recursively mark its car and cdr.
5. Otherwise, the object is a gvector. Recursively mark its elements.

Marking an object thus involves ensuring that its mark bits are set and then recursively marking any pointers contained within the object if the object was originally unmarked. If this recursive step was implemented in the obvious manner, marking an object would take stack space proportional to the length of the poiner chain from some root to that object. Rather than storing that pointer chain implicitly on the stack (in a series of recursive calls to the mark subroutine), the PPCCL marker uses a technique called *link inversion* to store the pointer chain in the objects themselves.[1]

As noted earlier, some roots (certain register values saved in exception frames and certain slots in control stack frames) are treated as locatives by the mark phase. If a root is a gvector locative and tagged as a fixnum and the root's markbit is clear, the marker must scan backward for the gvector's header and mark the gvector as usual. If the root is a pc locative and tagged as a fixnum and otherwise unmarked, the mark phase must scan backward for the code vector's header and mark the code vector as usual. Typically, objects pointed "into" by locatives are also pointed "to" by some other pointers and the marker usually traverses non-locative roots before traversing locative roots, so the overhead involved in this backward scan is usually avoided.

Certain types of objects are treated a little specially:

1. To support a feature called GCTWA[2], the vector which contains the internal symbols of the current package is marked on entry to the mark

[1] The "naive" recursive technique may be faster. A hybrid scheme which uses recursion when sufficient stack space is available and falls back to link inversion otherwise is also possible. Profiling has usually indicated that most GC time is spent in later phases, so this optimization has not yet been implemented in PPCCL.

[2] I believe that the acronym comes from MACLISP, where it stood for "Garbage Collect Truly Worthless Atoms".

phase but the symbols themselves are not marked at this time. Near the end of the mark phase, symbols referenced from this vector which are not otherwise marked are marked if and only if they're somehow distinguishable from newly created symbols (by virtue of their having function bindings, value bindings, plists, or other attributes.)

2. POOLs have their first element set to NIL before any other elements are marked.

3. All hash tables have certain fields (used to cache previous results) invalidated.

4. WEAK HASH TABLEs and other weak objects are put on a linked list as they're encountered; their contents are only retained if there are other (non-weak) references to them.

At the end of the mark phase, the markbits of all objects which are transitively reachable from the roots are set and all other markbits are clear.

5.2 Relocation phase.

The *forwarding address* of a doubleword in the dynamic heap is (<its current address> - (8 * <the number of unmarked markbits that precede it>)) or alternately (<the base of the heap> + (8 * <the number of marked markbits that preced it>)). Rather than count the number of preceding markbits each time, the relocation table is used to precompute an approximation of the forwarding addresses for all doublewords. Given this approximate address and a pointer into the markbits vector, it's relatively easy to compute the exact forwarding address.

The relocation table contains the forwarding addresses of each *pagelet*, where a pagelet is 256 bytes (or 32 doublewords). The forwarding address of the first pagelet is the base of the heap. The forwarding address of the second pagelet is the sum of the forwarding address of the first and 8 bytes for each mark bit set in the first 32-bit word in the markbits table. The last entry in the relocation table contains the forwarding address that the freepointer would have, e.g., the new value of the freepointer after compaction.

In many programs, old objects rarely become garbage and new objects often do. When building the relocation table, the relocation phase notes the address of the first unmarked object in the dynamic heap. Only the area of the heap between the first unmarked object and the freepointer needs to be compacted; only pointers to this area will need to be forwarded (the forwarding address of all other pointers to the dynamic heap is the address of that pointer.) Often, the first unmarked object is much nearer the free pointer than it is to the base of the heap.

5.3 Forwarding phase.

The forwarding phase traverses all roots and the "old" part of the dynamic heap (the part between the base of the heap and the first unmarked object.) All references to objects whose address is between the first unmarked object and the free pointer are updated to point to the address the object will have after compaction by using the relocation table and the markbits vector and interpolating.

The relocation table entry for the pagelet nearest the object is found. If the pagelet's address is less than the object's address, the number of set markbits that precede the object on the pagelet is used to determine the object's address; otherwise, the number of set markbits the follow the object on the pagelet is used.

Since forwarding views the heap as a set of doublewords, locatives are (mostly) treated like any other pointers. (The basic difference is that locatives may appear to be tagged as fixnums, in which case they're treated as word-aligned pointers into the object.)

If the forward phase changes the address of any hash table key in a hash table that hashes by address (e.g., an EQ hash table), it sets a bit in the hash table's header. The hash table code will rehash the hash table's contents if it tries to do a lookup on a key in such a table.

Profiling reveals that about half of the total time spent in the GC is spent in the subroutine which determines a pointer's forwarding address. Exploiting GCC-specific idioms, hand-coding the routine, and inlining calls to it could all be expected to improve GC performance.

5.4 Compact phase

The compact phase compacts the area between the first unmarked object and the freepointer so that it contains only marked objects. While doing so, it forwards any pointers it finds in the objects it copies.

When the compact phase is finished, so is the GC (more or less): the free pointer and some other data structures are updated and control returns to the exception handler that invoked the GC. If sufficient memory has been freed to satisfy any allocation request that may have triggered the GC, the exception handler returns; otherwise, a "seriously low on memory" condition is signalled, possibly after releasing a small emergency pool of memory.

6 The ephemeral GC.

In the PPCCL memory management scheme, the relative age of two objects in the dynamic heap can be determined by their addresses: if addresses X and Y are both addresses in the dynamic heap, X is younger than Y (X was created more recently than Y) if it is nearer to the free pointer (and farther from the base of the heap) than Y.

Ephemeral (or generational) garbage collectors attempt to exploit the following assumptions:

- most newly created objects become garbage soon after they're created.
- most objects that have already survived several GCs are unlikely to ever become garbage.
- old objects can only point to newer objects as the result of a destructive modification (e.g., via SETF.)

By concentrating its efforts on (frequently and quickly) reclaiming newly created garbage, an ephemeral collector hopes to postpone the more costly full GC as long as possible. It's important to note that most programs create **some** long-lived garbage, so an EGC can't typically eliminate the need for full GC.

An EGC views each object in the heap as belonging to exactly one *generation*; generations are sets of objects that are related to each other by age: some generation is the youngest, some the oldest, and there's an age relationship between any intervening generations. Objects are typically assigned to the youngest generation when first allocated; any object that has survived some number of GCs in its current generation is promoted (or *tenured*) into an older generation.

When a generation is GCed, the roots consist of the stacks, registers, and global variables as always and also of any pointers to objects in that generation from other generations. To avoid the need to scan those (often large) other generations looking for such intergenerational references, the runtime system must note all such intergenerational references at the point where they're created (via SETF).[3] The set of pointers that may contain intergenerational references is sometimes called *the remembered set.*

In PPCCL's EGC, the heap is organized exactly the same as otherwise; "generations" are merely structures which contain pointers to regions of the heap (which is already ordered by age.) When a generation needs to be GCed, any younger generation is incorporated into it; all objects which survive a GC of a given generation are promoted into the next older generation. The only intergenerational references that can exist are therefore those where an old object is modified to contain a pointer to a new object.

The EGC uses exactly the same code as the full GC. When a given GC is "ephemeral",

- the "base of the heap" used to determine an object's markbit address is the base of the generation being collected;
- the markbits vector is actually a pointer into the middle of the global markbits table; preceding entries in this table are used to note double-word addresses in older generations that (may) contain intergenerational references;

[3]This is sometimes called "The Write Barrier": all assignments which might result in intergenerational references must be noted, as if the other generations were write-protected.

- some steps (notably GCTWA and the handling of weak objects) are not performed;
- the intergenerational references table is used to find additional roots for the mark and forward phases. If a bit is set in the intergenerational references table, that means that the corresponding doubleword (in some "old" generation, in some "earlier" part of the heap) may have had a pointer to an object in a younger generation stored into it.

The intergenerational references table is maintained indirectly: whenever a SETF operation that may introduce an intergenerational reference occurs, a pointer to the doubleword being stored into is pushed onto the *memo buffer*, which is a stack whos top is addressed by the memo register. Whenever the memo buffer overflows[4] when the EGC is active, the handler scans the buffer and sets bits in the intergenerational references table for each doubleword address it finds in the buffer that belongs to some generation other than the youngest; the same scan is performed on entry to any ephemeral GC. After (possibly) performing this scan, the handler resets the memo register to point to the bottom of the memo stack; this means that when the EGC is inactive, the memo buffer is constantly being filled and emptied for no apparent reason.

With one exception (the implicit SETFs that occur on entry to and exit from the binding of a special variable), all SETFs that might introduce an intergenerational reference must be memoized.[5] It's always safe to push any cons cell or gvector locative onto the memo stack; it's never safe to push anything else.

Typically, the intergenerational references bitvector is sparse: a relatively small number of old locations are stored into, although some of them may have been stored into many times. The routine that scans the memoization buffer does a lot of work and usually does it fairly often; it uses a simple, brute-force method but might run faster if it was smarter about recognizing addresses that it'd already seen.

When the EGC mark and forward phases scan the intergenerational reference bits, they can clear any bits that denote doublewords that definitely do not contain intergenerational references.

7 Source file organization

The CCL directory hierarchy is organized into the following subdirectories:

ccl:compiler; (lisp source to the compiler proper - not COMPILE-FILE)

ccl:compiler;PPC (lisp source to the ppc backend)

ccl:compiler;sparc (lisp source to the sparc backend)

[4] A guard page at the end of the memo buffer simplifies overflow detection.

[5] Note that the implicit SETFs that occur when initializing an object - as in the case of a call to CONS or VECTOR - can't introduce intergenerational references, since the newly created object is always younger than the objects used to initialize it.

ccl:level-0; (lisp source to the bootstrapping image)

level-0;PPC; (ppc-specific lisp sources used to create the bootstrapping image)

level-0;sparc; (sparc-specific lisp sources used to create the bootstrapping image)

ccl:level-1; (lower-level lisp sources)

ccl:lib; (nominally higher-level lisp sources, macros, etc.)

ccl:library; (originally "things used to build lisp but not part of image")

ccl:lisp-kernel; (C and PPC assembler sources to the kernel)

ccl:lisp-kernel;linux; (linux-specific kernel build directory)

ccl:lisp-kernel;vxworks; (vxworks-specific kernel build directory)

ccl:sparc-kernel; (C and SPARC assembler sources to the kernel)

ccl:xdump; the bootstrap image loader

8 Build process

8.1 Kernel build process

The linux kernel can be built under PPC using "mostly" standard Linux/GNU development tools:

- cc - the C compiler, both egcs and gcc have been used
- ld - the GNU linker
- m4 - the GNU m4 macro processor
- gas- a slightly modified version of GNU as from the GNU binutils 2.9.1 release[6]

The kernel sources consist of the sources to the kernel proper and the "subprims"; there's very little direct interaction between the subprims and the rest of the kernel, and they can be built as a seperate object module by changing a few lines in the Makefile.

[6]As it was released, GNU as didn't generate line number info correctly when assembler source files contained more than one function definition. I had to add support for a new pseudo-op in order to get this to work. All of the assembler sources in the kernel include a file of m4 macros which may expand into this pseudo-op; to assembler the kernel sources using an unpatched version of GNU as, those macros would need to be changed.

Having correct line-number information in the object file makes source-level debugging of the assembly-language code possible.

The subprims are assembly-language routines called by compiled lisp code. The subprims object code starts with a jump table; a given subprimitive has a known (to the compiler) and constant jump-table index associated with it.[7] The table is assumed (there's bound to be code somewhere that makes this assumption) to contain 256 entries; about 100 entries are actually defined.

If the subprims module is linked separately, it imports nothing from any other module (all of the data it accesses is presumed to be relative to RNIL), and it only needs to export the address of the "start_lisp" routine. "start_lisp" is not itself a subprim: it's called from the kernel to ... start running lisp code, of all things, and needs to be able to make pc-relative calls to other routines in the subprims code to do so.

The subprims module is generated from the following source files (all of them in the "ccl:lisp-kernel" directory)

sp_jump.s the jump table itself; the corresponding .o file should always be listed before other subprims files when linking

sp_bind.s subprims to effect special (dynamic) binding

sp_builtin.s subprims to simplify calls to frequently-called lisp functions

sp_call.s procedure call, function entry & exit.

sp_catch.s catch, throw, unwind-protect

sp_ffi.s foreign function call, callback

sp_heap_cons.s heap allocation

sp_lambda.s runtime support for &optional, &key, &rest ...

sp_stack_cons.s stack allocation

sp_values.s multiple-value accepting/returning

sp_vref.s uvref/(setf uvref)

sp_end.s marks end of subprims; should be linked last

The kernel proper is generated from the following C and assembler sources:

bits.c bitvector routines

gc.c the garbage collector

[7] The compiler emits subprimitive calls as absolute branches to this index. Before a code vector can be executed, the system has to fix up all subprimitive calls that it contains, by adding the runtime value of the start of the subprims jump table to the target of each absolute branch instruction. When code is saved (e.g., to a .pfsl file or heap image), subprim calls have to be "normalized", since the runtime address of the base of the subprims jump table may vary from invocation to invocation.

kernel-init.c some low-level OS support

lisp-exceptions.c exception handling

loader.c heap image loader

mathtrap.c fixnum overflow handlers

pef.c PEF (Macintosh executable format) loader, image dumper

plbt.c debugging support: backtrace

lispdcmd.c debugging support: utilities

plprint.c debugging support: lisp object printing

plsym.c debugging support: find symbol matching pname

pmcl-kernel.c startup code ("main()")

thread_manager.c cooperative thread creation, context switch

vxlow.c VxWorks support

asmutils.s cache maintenance and other assembly language routines called from kernel

imports.s maintain "kernel imports" table: C funcs called from lisp code.

qtppc.s thread context switch

puppet.c native scheduler interface

malloc.c a public-domain malloc implementation

Assuming that all of the required tools are installed (e.g., the modified "gas") and on the search path, building the kernel image on a PPC Linux host involves changing to the appropriate target directory ("ccl:lisp-kernel;linux;" or "ccl:lisp-kernel;vxworks;") and typing "make". The linux kernel will wind up in the file "ccl:ppccl"; the vxworks kernel will wind up in "ccl:vxppccl.o".

It's been a while since I've cross-compiled a VxWorks kernel from PPC Linux; the makefile is now set up to use the VxWorks gcc tool chain (and the modified gas.) Some versions of gcc/egcs under PPC Linux can't generate dwarf debugging output, which is the debugging format preferred by gdb under VxWorks, so it's actually preferable to use the VxWorks/GNU tools.

The vxworks and linux kernels are conceptually identical but differ in some implementation details.

In the vxworks kernel, the subprimitive files are linked into a relocatale object module ("lisp_subprims.o") that's loaded when the (rest of) the kernel is first initialized. (This is done to ensure that the subprimitive entry points are loaded into the low 32MB of memory; the same thing is done to a module called "vxlow.o", which provides exception-handling support under VxWorks.)

The relocatable images produced by the linker are post-processed by a tool "unrelppc" which removes PC-relative branches to external addresses and generates code sequences to do "far calls" to those addresses.

When the PPCCL image starts up, it makes a call to the OS's malloc() to obtain a large block of memory; the kernel then uses its own malloc implementation to allocate lisp's heap and its own data structures. When the application exits, a few global variables are reset and (under VxWorks) the same large block can be reused.

8.2 Lisp build process.

Building an MCL image basically requires a working MCL; it's built in a couple of stages.

A "bootstrapping image" - typically named "ccl:ccl;ppc-boot" is loaded by the kernel. It contains:

- the fasloader
- the guts of the symbol-lookup and interning code used by the fasloader.
- most of the code which implements hash-tables
- a lot of arithmetic code
- a lot of random pieces of functionality that are hard to bootstrap

All told, ppc-boot is around 500Kb in size. It could probably be made smaller, but the general philosophy has been to throw anything that might be useful and hard to bootstrap into that initial image. When the ppc-boot image is entered, it calls a few functions to initialize data that couldn't be initialized by other mechanisms and then calls:

(ccl::%fasload "level-1.pfsl")

"level-1.pfsl" contains calls to %FASLOAD several dozen other files. A flag is set that causes the fasloader to write each filename to file descriptor 2 before trying to load it; this can sometimes help to isolate bugs (especially those that occur early in the loading sequence, before lisp can do its own I/O.)

%FASLOAD returns NIL (rather than signalling an error) if it encounters a problem. When the bootstrapping flag is set, it prints a message to that effect if it's unable to load a file. (Early in the loading sequence - before ERROR is defined, for instance - it's hard to signal a lisp error.)

When %FASLOAD finishes loading a few dozen files, MCL enters a read-eval-print loop.

For some weird reason (having to do with how "ppc-boot" is made), it's not possible to enable the EGC in this environment, though just about everything else that would work in a full VxWorks/Linux image should work. As a matter of fact, at this point the typical thing to do is to call SAVE-APPLICATION to make a new heap image. ppc-boot has done its job ...

"ppc-boot" is produced from the .pfsl files in the "ccl:level-0;" directory; those .pfsl files are produced from the .lisp source files in that directory. Calling:

```
? (ccl::xload-level-0)
```

will recompile any files in the level-0 directory that are out-of-date, then load the .pfsl files in that directory into a "simulated lisp heap" that's allocated inside a large vector in the lisp. When all of those files have been successfully loaded, the "simulated heap" is written to "ccl:ccl;ppc-boot".

"Cross-dumping" a ppc-boot image takes a fair amount of memory; running the ppc-boot image requires more memory than running the equivalent saved image will. In an 8MB partition (the default), there's enough free space to save a new ppc-boot but it might get a little tight if it's necessary to recompile any of the level-0 sources. It's generally better to use a saved MCL image to cross-dump from.

There's a very ancient set of utilities for recompiling the "higher-level" MCL sources. It tries to pay attention to which files can and can not be loaded more than once (because of bootstrapping screws.) Calling:

```
? (ccl::compile-ccl)
```

will recompile lisp files that are out-of-date and load some of them after they've been recompiled;

```
? (ccl::compile-ccl t)
```

will force recompilation of all lisp sources.

CCL::XCOMPILE is analogous, but doesn't load any of the files that it may have recompiled.

Whether or not one loads recompiled files back into the environment is usually a matter of preference: doing so takes longer and uses more memory, but may help to catch simple typos or warnings. Sometimes, changes to macros or constants may imply that the changed file be loaded into memory before its dependants; sometimes, changes require that that -not- happen.

Calling ([X]compile-ccl t) will produce a lot of warnings, most of which come from code that isn't executed. They should be cleaned up someday, if only because they tend to obscure "real", legitimate warnings that may apply to newly-changed code.

In general, an image intended for production use should not have been compiled in. SAVE-APPLICATION saves code-vectors and symbol pnames to a "pure" (conceptually read-only) heap area, under the assumption that those objects are permanent and that it's better to just keep them out of the GC's way. There's no way to un-purify something once it's been purified, so doing something like (COMPILE-CCL T) will result in two copies of most code vectors being saved to the new pure section, even though one of them may have become unreachable.

There are some differences between the way MacOS MCL's pathname-parsing code works and the way that the VxWorks/Linux port's pathname code works; in the latter case, "home:init.lisp" isn't handled the same way as it is in the former. Until this is fixed, it's wise to call SAVE-APPLICATION with an :INIT-FILE argument of NIL.

9 Exceptions.

MCL programs generate exceptions; the MCL kernel installs one or more handlers for these exceptions using whatever facilities are available in the OS.

In general, the assumption is that it's both simpler and faster to allow an exceptional case to generate an exception than it is to check for that case. There are some scenarios which might violate that assumption, but I believe that it's true in general.

MCL programs generate illegal-instruction exceptions, trap exceptions, and write-protect exceptions. These exceptions map (more-or-less) to POSIX signals, so the exception handlers are defined as POSIX signal handlers. The POSIX signal facilities in VxWorks seem not to work very well, so vxmcl.o installs its own low-level handlers (in the file "vxlow.c/.o") that map hardware exceptions to POSIX signals and use a signal-like interface to call user-defined handlers.

In PPC Linux, a signal handler receieves as a second argument a pointer to a "sigcontext" structure, which describes the machine state at the time of the exception and other information. The VxWorks exception handlers generate a (supposedly) identical structure, so most of the MCL kernel and lisp code for exception handling doesn't have to be conditionalized between those platforms.

9.1 UUOs (illegal instructions).

Certain PPC opcodes are defined as UUOs (the term came from early DEC PDP architectures where the acronym stood for "Unimplemented User Operation". They can be thought of as extended (and expensive) PPC instructions that are executed by an interpreter in the MCL kernel. Many of the defined UUOs are used to signal errors (so the fact that this is a rather expensive operation isn't usually very significant.) A few UUOs are used to handle cases where a fixnum arithmetic operation has generated an overflow (and the result must be stored in a bignum.) Handling this case inline would add 6-8 instructions to every compiled instance of (+ (the fixnum x) (the fixnum y)); doing it inline is a lot faster than handling and decoding an exception is.

The DISASSEMBLE function recognizes UUOs and prints them (and their operands) symbolically.

9.2 Traps.

Many Lisp runtime errors are tested for (and signalled) by use of PPC "TW" and "TWI" instructions. For instance, a function that takes 0 arguments and wants to ensure that it received 0 arguments might have as its first instruction:

```
(twnei nargs 0)
```

A handler for this has enough information available (from decoding the instruction) to be able to signal a "wrong-number-of-arguments" error. Trap instructions are also used for type- and bounds- checking, and the handler may have

to look at preceding instructions in order to decode the trap. For instance, in the code fragment:

```
;; Ensure that arg_x contains a simple-base-string. Use imm0 as a
;; temporary.
clrlwi imm0,arg_x,30        ; extract lisptag of arg_x to imm0
cmpwi imm0,ppc::tag-misc    ; object with subtag ?
bne @1                      ;[8]
lbz imm0,ppc::misc-subtag-offset(arg_x)
@1 twnei imm0,ppc::subtag-simple-base-string
```

the trap handler has to be able to deduce that imm0 was initialized from arg_x, and should therefore report that arg_x has the wrong type. (The trap handler deduces this by looking at the preceding instructions: it should not be possible to reach the lable @1 in the code above without having executed the other instructions in that sequence.)

Both UUOs (those used to signal errors) and TRAPs call out to lisp code (through the value cell of a "nilreg-relative-symbol") in order to actually signal the error. The lisp code that's called has to be able to interpret the sigcontext structure's fields (and may have to decode some instructions) in order to generate a meaningful error. If the lisp callback function isn't initialized (as will be the case when a bootstrapping image is loading or when a saved image is first initializing itself), the kernel tries to decode the exception itself, prints a message, and enters a kernel-level debugging loop.

9.3 Write-protection.

The kernel allocates write-protected pages at the "end" of various data region (stacks, the dynamic heap, and the EGC memoization buffer.) For every write-protected region, it maintains a data structure which describes the range of pages that are write-protected and some information about why they're protected. On receiving a possible write-protection exception (usually mapped to SIGSEGV), the MCL kernel's exception handler determines whether the exception was caused by a write and, if so, what address was being written to.[9] . It then does a linear search of its list of protected areas until it finds a protected area which contains the target adress and, if it finds one, calls a handler associated with that "protection reason." If that handler indicates that it was able to successfully resolve things, the exception is returned from and execution resumes; otherwise, a kernel-level debugger is invoked.

There are several "protection reasons":

[8] Could also have trapped here, but the "expected type" wouldn't be very informative.

[9] Memory-access errors not caused by writes aren't handled by MCL

9.3.1 tstack, vstack guard pages.

Recall that each MCL thread has three stacks (control, temp, value). Each of these stacks is composed of one or more *segments*, and each segment of the tstack and vstack has a protected guard page at the physically lowest address in the segment.[10] When a segment overflows, a new segment is allocated and (after lots of incredibly hairy code) made current; the top few frames on the old segment are copied to the new segment and pointers are fixed up. Magically, when those frames exit, the old stack segment is restored and the new one is marked as "empty". If the old segment overflows again, the handler will find that a younger segment already exists so much of the overhead can be avoided.

The kernel calls out to lisp (actually LAP) code to handle these overflows. If the total size of all segments in a lisp thread's stacks exceeds a threshold, a continuable error is signalled which offers the option of raising that threshold and continuing or aborting a possible case of runaway recursion.[11]

The code that handles these overflows is incredibly hairy (what happens if one stack overflows when you're trying to fix a case where the other one overflowed?) and seems to be very robust. Bill St. Clair wrote most of it and I'm not too familiar with it in detail.

It's hard to know what an "optimal" segment size would be: too large would defeat the purpose of having segmented stacks and too small would cause overflow traps to occur very frequently. I believe that the default is usually set to somewhere between 16K and 32K bytes (+ 4K for the guard page.)

9.3.2 memoization buffer.

As mentioned above, SETFs that could introduce an intergenerational reference are noted by pushing a pointer to the doubleword being stored into onto a "memoization buffer". When this buffer overflows (when an attempt is made to store into a guard page at the end of the buffer), the entries in the buffer are processed (if the EGC is active) and the register cointaining the current memo buffer pointer is reset to point to the beginning of the buffer.

The memoization buffer is 32KB (8K entries) long.

9.3.3 heap guard segments.

The kernel memory management code views the heap as consisting of consisting of a number of *segments*, where each segment is 32KB (8 4KB pages). These segments are aligned relative to the end of the dynamic heap; the size of the heap is generally not a multiple of the segment size, but that doesn't matter. The

[10] The control stack doesn't use guard pages for overflow detection: on many platforms, exception information is pushed on the control stack by the OS. If the SP register is at or near a protected page, that'll cause a write-protect exception and the OS will try to push an exception frame on the control stack.

[11] The code compares the allocated size of the thread's stack segments to the threshold; arguably, it'd be more meaningful to look at the sum of the threads "active" stack segment usage.

first segment beyond the free pointer is write-protected, and memory between the free pointer and that protected segment is zeroed.

When lisp code is running (and not in the act of initializing a newly-allocated object), two registers (freeptr and initptr) point to the free pointer.

An object's *physical size* is (the sum of its logical size in bytes + (4 if the object has a header, 0 otherwise), rounded up to a multiple of 8. The physical size of a cons cell is thus 8, the physical size of a 3-character simple-base-string is also 8, and the physical size of a 5-character simple-base-string is 16. Any object whose physical size is <= 32KB can be allocated without concern for heap-size limitations; any object whose physical size is a constant < 32KB can be allocated in a single instruction.[12] That instruction is one which involves a "store with update" of rzero to an offset from the freeptr; this type of instruction atomically writes rzero to the new free pointer and advances the freeptr to point to the new free pointer (the initptr register is left pointing at the old free pointer, and can be used to initialize the object.) After the object's been initialized, the initptr is then set to point to the new free pointer.

For example, to set arg_z to (cons arg_y nil), one would say:

```
(stwu rzero ppc::cons.size freeptr)  ; advance the freeptr
(la arg_z ppc::fulltag-cons initptr)
(mr initptr freeptr)
(stw arg_y ppc::cons.car arg_z)
(stw rnil ppc::cons.cdr arg_z)
```

The compiler actually generates slightly different code for this case, but the code above is maybe a little clearer. The GC assumes that whenever it can run, the freeptr and initptr are equal; if there were ever a preemptive scheduler, the GC would have to recognize cases where they were different and interpret that to mean that an object was half-initialized. There are places in compiled and LAP code where the initptr is used to initialize some other (possibly stack-allocated) object; that idiom is supposed to help the GC recognize that the stack-allocated object is in the process of being initialized.

Newly-allocated heap memory is always guaranteed to be zeroed; in the code above, the initptr was set to the freeptr as soon as it had been used to set the result.

Memory allocation where the physical size is not a compile-time constant or is >= 32KB is handled by a subprimitive call. If the subprim determines that the physical size of the object is <= 32KB, it sets a temporary register to that size and does:

```
stwux rzero,freeptr,temp
```

[12] Until recently, there was a bug whereby the compiler would attempt to allocate objects whose physical size was exactly 32KB inline. It was correct in thinking that that "should work", but the constant was unfortunately sign-extended ...

If the size is found to be > 32KB, a UUO is used to trap into the kernel; the UUO handler tries to "manually" advance the freepointer, manipulate guard pages, and possibly force a GC to allocate the large object. (The last time I checked, there was exactly one object in MCL whose size was >32KB.)

Some of the time, the STWU[X] instructions will write to a protected page and invoke the MCL kernel's protection handler.

If the protected page is in the last heap segment, a full GC is invoked. If after GC the freepointer would still be in the last segment, a "chronically out of memory" condition is signalled. Otherwise, the GC has updated the freeptr and initptr registers, so returning from the exception will cause the allocation attempt to be retried & it'll succeed.

If the protected page was in some segment other than the last, the handler will either:

- invoke the EGC if the size of the youngest generation exceeds a specified threshold, or
- simply unprotect the segment, zero its contents, protect the next segment, and return from the exception.

Zeroing a segment happens very quickly (by means of a sequence of 'dcbz" instructions, each of which zeros an entire 32-byte cache line.

10 Heap images.

Under MacOS, the output of a call to SAVE-APPLICATION is an executable file in Apple's PEF executable file format, or, more accurately, a PEF image followed optionally by compressed lisp heap data. Lisp programs tend to contain much more initialized data than C programs do and a large percentage of that data consists of pointers to other objects. Since the target OS doesn't guarantee that programs will be loaded at consistent base addresses[13], that data must be relocated at load time. The mechanisms available for noting relocation information in object file formats like PEF and ELF don't really scale well, and tend to be overwhelmed by the relocation requirements of MCL's heap.

Since MCL's tagging scheme allows it to reliably determine which objects are pointers, it doesn't need auxiliary information to determine which objects need to be relocated; since lisp objects (at least those that can be preserved in a saved image) can only point to other lisp objects and since lisp objects can only be allocated in certain areas (the static, dynamic, and readonly heaps, in general) all that's needed in order to relocate a lisp object reference is to know where those areas were when the image was saved and where they are (or will be) when the image is loaded. As it happens, there are more concise ways of encoding many types of lisp data references, so the process of encoding these references allows for a simple but effective compression scheme.

[13]Linux does, but different Linux releases choose different defaults.

PEF is a reasonable way of encoding the size and position of the image's "pure" section; a small area of "the static heap" - a few hundred bytes on either side of NIL - is described as a PEF data section (with a few pointers embedded in it that use PEF relocation mechanisms.) In the case of the "ppc-boot" image, the static heap "is all mutable data in that image"; again, a PEF data section is used to contain the image data and PEF relocation information is used to describe how data references should be resolved. If the image file is larger than what's described in the PEF header, the remaining bytes are assumed to contain a header (which indicates the addresss ranges where lisp objects were allocated when the image was saved) followed by a compressed representation of those objects.

Since an object's type is encoded in its low bits, an object that can't be represented more concisely is stored in a form where the least-significant byte comes first.[14] The code in the image loader that reads compressed lisp data does something like:

```
read-a-byte-from image-file
if (is-a-special-operator byte)
  handle-special-case
else
  read-three-more-bytes, form word, rotate it around
  if that word is tagged as a pointer
    lookup the area containing that pointer in the table
    determine where the corresponding area is in the "new" image
    subtract the base of the old area from the pointer
    add the base of the new area to the pointer
  endif
  store that word in memory
endif
```

The special cases involve encoding references to NIL, small fixnums, "nearby" objects, etc. concisely.

This scheme usually compresses the "data section" by about 40% and costs very little; if the heap were stored uncompressed and read from the image file in a single I/O operation, it would still need to be relocated a word at a time. The special cases are much faster than the general case outlined above and they're also pretty common.

11 Operating system dependencies.

11.1 PPC Linux.

PPCCL requires version 2.2.14pre6 or later of the Linux kernel and version 2.1.3-0b or later of the GNU C library (glibc). Earlier versions of the kernel and C library had severe thread-related bugs.

[14] It isn't truly little-endian; it's just rotated around.

I'm not sure whether appropriate versions of these files are available as part of the "LinuxPPC 2000" distribution from www.ppclinux.com; the files are available from www.linuxppc.org. Since Java also uses Native Threads, it'd be a good idea to track the web page at <http://www.linuxppc.org/java>; whatever minumum kernel/glibc versions Java requires are likely to be required by ppccl as well.

11.2 PPC VxWorks.

I've run vxppccl.o under both the "wind-2.0" and "wind-1.x" versions of VxWorks; I've never had any luck getting a debugger to work under 2.0.

VxWorks is missing some POSIX-y functionality; workarounds are probably available, but aren't implemented:

- rename() is missing. Presumably, RENAME-FILE would have to call the host shell to rename a file.
- ftruncate() is missing.[15] If I recall correctly, SAVE-APPLICATION fails if the output file exists because of this bug. I don't think that COMPILE-FILE is affected.
- filesystem links don't work: one can find out (via stat()) that a file is a link, but there's no way to resolve it (i.e., readlink() doesn't exist.) This means that functions like TRUENAME can sometimes give the wrong answer.
- there isn't a general mechanism for emulating an /etc/passwd file: USER-HOMEDIR-PATHNAME will always return "/home/byers". That's rarely the correct answer.
- there are a number of cases where the VxWorks kernel "wraps" a Linux-like interface around a VxWorks function. This is intended to hide the implementation differences between Linux and VxWorks from the lisp filesytem code. The good news is that there aren't too many such differences; the bad news is that the code that tries to hide them when they exist can be a little obscure.

12 Threads and the scheduler.

A *stack group* is a related set of control, value, and temporary stacks. For historical reasons (having to do with LispM compatibility), a stack group is a subtype of FUNCTION. One can *preset* a stack group (giving it an initial function and arguments) and one can *resume* a stack group (causing it to begin or continue execution.) Funcalling a stack group is essentially equivalent to

[15] ftruncate() is used to set the length of an open file. It can be optionally linked into VxWorks, but it doesn't work.

resuming it. A stack group whose initial function has returned is said to be *exhausted*; it is an error to attempt to resume an exhausted stack group.

Each stack group maintains its own set of special bindings.

A *lisp process* is an object which contains an initial stack group and scheduling information (priority, etc.) Since stack groups are first-class objects, it's possible (though unusual) for a process's current stack group to be something other than its initial stack group.

At any point in time, one process (the value of CCL::*CURRENT-PROCESS*) and one stack group (CCL::*CURRENT-STACK-GROUP*) are current or active.

Activating a process (other than the current process) is basically just a matter of resuming its current stack group and updating *CURRENT-PROCESS*.

Resuming a stack group involves ... lots of code. Any stack group that's not current looks like it's just executed a call to a foreign function that saves and restores the stack group's registers. (If the stack group isn't newly created, this is in fact what it's just done.) The code which does a stack group resume is conceptually something like:

```
(undo-special-bindings)
(ff-call "yield-to-thread" next-stack-group)
(redo-special-bindings)
```

UNDO-SPECIAL-BINDINGs undoes special bindings (in reverse order, most recent first.) REDO-SPECIAL-BINDINGs reestablishes the stack-group's special bindings (most recent last.)

MCL's scheduler has historically been "pseudo-preemptive": the compiler generates code which tests the value of the global variable CCL::*INTERRUPT-LEVEL* and traps if that value is positive on entry to most functions and at the heads of most loops. A timer interrupt or some other asynchronous event sets that variable; lisp code tests it synchronously and at times when it's safe (gc-wise) to schedule another process or do other periodic activity.

Under MacOS, the periodic interrupt was originally used to poll for UI events, and the instruction sequence that tests *INTERRUPT-LEVEL* is sometimes still referred to as "event polling."

CCL::*INTERRUPT-LEVEL* is a "nilreg-relative symbol" (a symbol allocated at a fixed offset from NIL and therefore accessable to the C code in the MCL kernel.) Its value is assumed to be a fixnum; when it's negative, lisp-level interrupts are disabled, when it's zero, they're allowed, and when it's positive, they're pending. Kernel code which tries to set *INTERRUPT-LEVEL* to signal that an interrupt is pending must do extra work if interrupts are disabled (via WITHOUT-INTERRUPTS) when it tries to do so. In this case, it must walk the linked list of saved special bindings on the current value stack until if finds a saved value of *INTERRUPT-LEVEL* that's non-negative; if it finds such a binding, it replaces it with a positive fixnum value, so that the interrupt will be pending when the outermost WITHOUT-INTERRUPTS is exited.[16]

[16]WITHOUT-INTERRUPTS is essentially just a macro that executes its body with

The act of Lisp code taking a trap in response to a pending interrupt is sometimes called a preemption, and the act of setting up a pending interrupt is sometimes called forcing a preemption. Usually, lisp will respond to a preemption request very soon after it's made, but because some code (e.g., GC) runs as if inside a WITHOUT-INTERRUPTS, it's hard to put an upper bound on this latency.

The latency between the time when an external event occurs and a preemption which schedules a lisp thread for execution in response can be forced can be minimized through the use of *proxy tasks*. A proxy task is an OS-level (preemptively scheduled) thread which acts "on behalf" of a corresponding lisp thread. There is a proxy task for every lisp thread that's created (including the initial lisp thread); there's also a distinguished OS thread called the lisp task.

When a proxy task is ready to run, it adds itself to a global queue (in priority order) and forces a preemption, then tries to go to sleep. When lisp responds to the preemption, it removes the highest-priority task from the proxy queue, makes the current processes proxy task runnable again, and activates the lisp task associated with the newly removed proxy task. If there was another task on the queue (of less or equal priority), that task is awakened so that it has a chance to force a preemption when the OS next schedules it. The lisp task's priority is always set to the priority of the proxy task whose lisp process is current.

A proxy task can block on behalf of its lisp process, waiting for a mutex, a counting semaphore, a select() call, or for a time period to elapse. When the lisp task wants the proxy for the current lisp process to block, it sets some fields in the proxy data structure, makes the underlying task runnable, then waits until some proxy appears on the proxy queue.

INTERRUPT-LEVEL bound to -1. The compiler treats it a little specially in that it avoids generating polling sequences for code inside a WITHOUT-INTERRUPTS form.