CMU Common Lisp Memory Layout under New Type Scheme

David B. McDonald

1991

Contents

1 Introduction

This document describes some ideas on how to layout memory under the new type scheme for CMU Common Lisp. There are many issues that are involved in this design. Each of the important issues that I have been able to identify is examined in a separate section. In each section two possible alternatives are proposed: one that assumes the default Unix 4.3BSD environment and one that assumes the significantly more advanced Mach environment. Throughout this document, I will use Unix to mean a non-Mach version of Unix, such as Berkeley 4.3BSD.

Section 2 describes the features of current Unix and Mach systems that affect the design. Section 3 describes a possible memory layout scheme under both Unix and Mach. Section 4 describes how the stacks should be layed out. Section 5 describes how memory could be allocated and managed with the new type scheme. Section 6 describes how garbage collection could work under the new type scheme assuming a simple stop and copy garbage collector. Section 7 describes how an initial kernel Lisp core file is generated. Section 8 describes how C code should be loaded into a running Lisp. Section 9 describes some idea on what the purification process should do to reduce the amount of work for the garbage collector. Section 10 describes how Lisp core files can be created so that the Lisp environment of a running process can be restored and run at a later time. Finally, section 11 summarizes what I believe is a reasonable approach to memory layout with the new type scheme.

2 Mach vs. Unix

Mach is a superset of Unix, so a design that relied on Mach without paying any attention to Unix would be more flexible and easier to implement. However, if CMU Common Lisp is going to run on non-Mach Unix systems, it is important to understand some of the short comings of these systems. In the following discussion, I am assuming a standard Unix 4.3BSD environment. Some vendor Unix systems, such as Dec's Ultrix or Sun's SunOs, may provide some or all of the features that make Mach a better operating to which to port Lisp. I don't have documentation on these, so don't know what if any features these systems support.

The following features are important to the design:

text segment The text segment contains the code for a process. This segment normally starts at address #x0. This is true for Mach. The important aspect about the text segment is that it is shared amongst all the process running the same program. For Lisp, in a Unix environment, this means it is important to have as much of the Lisp code and read-only data in this segment as possible to allow sharing. This segment is protected read-only by the operating system—writing to it is not allowed and will cause a segment violation. There is a separate part of an a.out file that contains the text segment.

data segment The data segment contains the data for a process. This segment normally starts at the beginning of the page after the end of the text segment. On the IBM RT PC it starts at #x10000000 instead. An a.out file contains two pieces of information for the data segment. The first is the size of the initialized data which is contained in the a.out file so that the data segment is initialized properly. The second piece of information is the size of the uninitialized data which when a program is loaded immediately follows the initialized data and is initialized to zero.

stack segment The stack segment contains the Unix user stack. This stack is normally near the high end of the address space and grows downwards toward the end of the data segment. An a.out files contains no information about the stack segment. Unix normally sets up the stack so that if it overflows, more memory is allocated automatically without the running process ever detecting anything. On the IBM RT PC, the stack segment starts at #xDFFFFFFC and grows downward.

memory allocation Memory allocation is where Unix and Mach diverge. On a Unix system, it is only possible to allocate memory at the end of the data segment by using the brk system call. There are subroutine libraries built on top of this that provide more flexible allocation. The memory allocated is always writable and there is no provision made to set the protection of this memory to no access (useful for guard pages of stacks and generational garbage collection) or read-only (useful for generation garbage collection). Note that any call out to C code could cause memory to be allocated by the brk system call either directly or through the subroutine libraries (e.g., malloc and free).

file mapping Unix does not currently provide any facility for mapping files into the address space of a process. A similar effect can be achieved by reading a file into memory. However, no sharing is possible as there is in Mach with its read-only or copy-on-write file mapping.

A convenient memory layout for the new CMU Common Lisp implementation would use the standard a.out executable file format and the brk memory allocation calls. This would allow CMU Common Lisp to run on less advanced versions of Unix than Mach. However, the Mach memory allocation and file mapping calls provide a degree of flexibility that make using the standard Unix calls, as they currently exist, difficult at best. Some of the current Unix BSD documentation contains definitions of unimplemented system calls that provide appropriate memory primitives such as allocation, protection, and file mapping that would allow Lisp to run on non-Mach machines without as much trouble.

3 Basic Memory Layout

The basic memory layout, assuming an a.out file format for Lisp save files follows (a similar layout could be used for Mach, although this is not necessary):

```
+-------------------------------------------------------+
|       C start up code                                 |
+-------------------------------------------------------+
|       Lisp Miscops (Lisp assembler)                   |
+-------------------------------------------------------+
|       Lisp read only data                             |
+-------------------------------------------------------+
|       Gap 1                                           |
```

```
+-------------------------------------------------------+
|       C data                                          |
+-------------------------------------------------------+
|       Miscop data                                     |
+-------------------------------------------------------+
|       Lisp static data                                |
+-------------------------------------------------------+
|       Gap 2                                           |
+-------------------------------------------------------+
|       Lisp dynamic 0 data                             |
+-------------------------------------------------------+
|       Lisp dynamic 1 data                             |
+-------------------------------------------------------+
|       Gap 3                                           |
+-------------------------------------------------------+
|       C stack                                         |
+-------------------------------------------------------+
```

The meaning of the above areas and what they contain is as follows:

C Start Up Code This area would correspond to the current lisp start up code. It would gain control from the operating system when lisp is run, pass in the environment and command line switches to Lisp, pick up the Lisp starting point from a known location and jump into Lisp code. This code would include the socket.o file that is necessary to open a connection to the X11 server.

Lisp Miscops This area would contain all the Lisp miscops including assembler routines and compiled miscops (e.g., bignum code).

Lisp Read Only Data This area would contain all the data in the default Lisp process that is read-only. This would include the combined function/code objects, documentation strings, symbol names, etc. that should never be modified.

Gap 1 There may be a region of memory between the end of the Lisp read only data and the beginning of C data. On the IBM RT PC, there is a gap since C code would start at #x0 and C data would start at #x10000000. On the Sun there isn't this separation between C code and data, so there may be no gap at all. The advantage of having a gap is that during the purification/save process for non system cores, read-only data could be moved into the Lisp read only area.

C Data This area contains the C data from the Lisp start up code.

Miscop Data This area contains any data areas the Lisp miscops need. For example, allocation information should be accessible from miscops. Allowing for miscop data requires enhancing the assembler to allow for resolving these addresses at miscop load time. The current miscops use a fixed location in memory to hold any data they need. This scheme is inappropriate on non Mach operating systems.

Lisp Static Data This area contains all the Lisp data from the default Lisp process that must be writable. This would include symbols, strings that are used as buffers, arrays, vectors, etc. In user saved Lisp files, this area should also include any C or foreign code loaded into Lisp.

Gap 2 A relatively small gap should exist, so that it is possible to allocate more data to static space without having to move dynamic space around.

Lisp Dynamic 0 Data This area is where Lisp starts allocating new dynamic objects when it first starts. A saved Lisp file may contain some initial data in this area.

Lisp Dynamic 1 Data The first GC will copy objects from the Lisp dynamic 0 data area to here and then flip back and forth between these two areas.

Gap 3 For both Mach and Unix, there will be a gap between the end of dynamic space and the C stack.

C Stack The C stack contains control information for the C start up code, including environment information and command line switches.

The important thing to notice about the above layout is that it is possible to use the Unix `a.out` file format to represent a Lisp save file. This is not important under Mach since file mapping is available and the various areas could lie anywhere in memory. However, on non-Mach systems there is far less flexibility on where these areas can be located.

The addresses, at which each of the areas described above start, depend on the particular machine and operating system. For example, on the IBM RT PC the C Start Up code starts at #x0, the Lisp Miscops would immediately follow, and then the Lisp Read Only Data. In Unix terminology these three areas would correspond to the text segment of an a.out file. This means they would be shared among all the processes running Lisp even on non Mach machines. The C Data starts at #x10000000, miscop data and the Lisp static data would immediately follow. In Unix terminology, this would correspond to the initialized data segment. The Lisp Dynamic 0 and 1 data areas should be allocated after this area. Note that if there is any dynamic data in the Lisp save file, it would follow the Lisp static area and be part of the initialized data segment. A gap of several pages should be left free between static and dynamic 0 data to allow static data to grow without causing a major garbage collection.

Note that nothing has been said about the location of the Lisp stacks. The following section describes stacks in more detail. The location of these stacks depend on whether Mach or Unix semantics are used.

4 Stacks

The new implementation of CMU Common Lisp appears to require the use of three stacks:

control stack The control stack contains information about the flow of control of a Lisp computation. This would include information about catch points for non-local exits, save areas for registers that must be preserved across function boundaries (including return points), and local variables that do not fit into registers. The control stack must contain only Lisp objects since it must be scanned by the garbage collector.

binding stack The binding stack contains binding information for special variables. As a computation progresses, special symbols and their old values are stored on this stack as these variables are bound. As a computation unwinds, this information is used to restore the old value to the symbol. The binding stack must contain only lisp objects since it must be scanned by the garbage collector.

non-lisp (or number) stack The non-lisp stack contains thirty-two bit objects that must never be processed by the garbage collector. This stack is necessary to allow writing portable number code in Lisp. I strongly recommend that the standard Unix stack be used for this purpose. This has no disadvantages that I can think of other than reserving the standard C register for this stack which on machines with thirty-two registers is not a problem. It has the advantage that system calls, calling out to C routines, and interrupt processing will be easier and more efficient. Also, in some Unix versions, the register containing the stack is treated specially and Lisp using it for something else may be dangerous.

There is one important consideration in the design of where these various stacks are located in memory. Pointers into any of the above three stacks will look like fixnums to the system, so it will be difficult to move the stacks around once they have been allocated. I believe there are two main avenues to follow here depending on whether Mach features are used or only those available in standard Unix.

Using Mach features, I believe the following stack scheme is the correct way to go. I am going to use stack locations that make sense for the IBM RT PC version of Mach. Other versions of Mach may use slightly different locations.

control stack The control stack should start at location #xBFFFFFFC and grow downward.

binding stack The binding stack should start at location #xCFFFFFFC and grow downward.

non-lisp stack The non-lisp or number stack should be the Unix stack and starts at #XDFFFFFFC and grows downward.

The advantage of this scheme is the stacks are separate so the GC needs no knowledge about the structure of the stacks. Also, they are at fixed locations and it is easy to restore a saved Lisp using the Mach `map_fd` call. Each stack can be up to 256 megabytes before overflow occurs. This should be large enough for some time to come. The control stack could actually grow bigger since it has only have dynamic memory below it and they will be separated by a large amount of space. Interrupt handling should be fairly straight forward since the initial interrupt information could automatically be stored on the Unix stack. An interrupt stack frame will be created on the Lisp control stack and any information that is necessary for Lisp can be copied from the Unix stack to the Lisp control stack.

For Unix, there are a couple of possible solutions to the stack problem. One solution is to put the control and binding stacks before or after static space. These stacks would be of fixed size and could not grow since it would be hard to move them because stack pointers would be difficult to find. To increase the size of the stacks, it would be necessary to rebuild the system. Also, the entirety of both stacks would have to be saved in a Lisp save file no matter if only one page was in use.

Another solution is to combine all of the stacks. This is probably the correct solution for Unix, but introduces significant complexities. Each Lisp stack frame would have to have enough information so that the Garbage collector could parse the frame. In particular, each frame would need to be divided into non-Lisp space and Lisp only space. Also, it would have to have a pointer to the previous active frame on the stack. Any stack information outside of a stack frame would be ignored by the garbage collector. A stack frame needs to specify which locations contain non-Lisp data and which contain Lisp data. The active frame pointer can be used to chase down the stack frames and GC the objects in the Lisp parts of each frame. This means the active frame pointer must always be valid when a GC occurs. Open but not active frames would be part of the frame for the function building the open frame. One problem is dealing with an arbitrary number of multiple values coming back from a function. If these come back on the stack, the Lisp/non-lisp information may need to be updated by the multiple-value return mechanism to make sure everything is in a legal state. The binding stack would use a mechanism similar to unwind-protect to restore old symbol values. Call out to C should be handled by using a frame on the stack that is outside the caller's stack frame and thus not part of the frame—GC will ignore it. Interrupts can be handled in one of two ways:

- An interrupt frame will be created on the stack and an interrupt handler called. While interrupts are disabled, the interrupt handler would be responsible for cleaning up the stack to make sure that it is consistent. It should copy the interrupt information that the Lisp interrupt processor needs and return from the interrupt causing control to pass to the real interrupt handler that can set up a legal Lisp frame (interrupts must still be disabled at this point).

- Have a separate interrupt stack as is done in the current system. A fixed location before or after the static region would be a likely candidate for this.

Saving the stack could be done by copying the contents into the `a.out` file just after the static space area. On restoring the file, it would be necessary to copy this information into the Unix stack (normally this will be a couple of pages).

5 Memory Management

Under Mach, using `vm_allocate` and `vm_deallocate` to allocate and free memory used for dynamic space is a big advantage. Lisp doesn't have to worry about where this memory is and new dynamic space will be zero-fill pages. This is much more efficient than under Unix where the new dynamic space (after the first GC) will contain old garbage and has to be paged in from disk. Under Mach, even if brk memory allocation is used, it will be wise to `vm_deallocate` and `vm_allocate` old space so that they are zero fill pages.

Under Unix, when Lisp starts up it must use the brk system call to allocate a chunk of memory, half of which will be used for dynamic 0; the other half for dynamic 1. The amount of space actually allocated should be specified by a command line switch. If dynamic memory usage grows sufficiently to require more memory to allocate, it will be necessary to allocate more memory to both dynamic spaces. However, it is not possible to guarantee that the new memory allocated will be contiguous with the top of dynamic 1 because C routines that have been called may have called the brk system call or more likely used a library call to allocate more memory.

It is convenient for the two dynamic memory areas to be the same size and each one to be contiguous. However, since under Unix there is no way to guarantee that the memory allocated with brk will be contiguous, a scheme that allows Lisp and non-Lisp memory to be intermingled may be necessary. A possible scheme is to have chunks of memory of some fixed size (e.g., 64K or 128K bytes) and do allocation in terms of these. This means the garbage collector and variable length allocation routines become more complicated. When one region of memory is exhausted there may be another that should be used.

Another possibility is to allow memory to expand only once effectively doubling the size of each dynamic space. This could be done by combining the original dynamic 0 and dynamic 1 space into dynamic 0 space and allocating a chunk of memory of the same size. Note a check could be made to see if the new memory will be contiguous and then Lisp could grow more slowly with doubling used as a last resort. It is also possible for Lisp to provide its own version of the C allocation calls. The only problem with this is that a C or assembler program could invoke the brk system call directly without going through the normal library routines.

6 Garbage Collection

Garbage collecting should be fairly straight forward. With two dynamic spaces, it is necessary only to copy the accessible objects from old dynamic space to new dynamic space.

Under Mach, when a GC is triggered, the following should be done:

1. Use `vm_allocate` to allocate at least as much space as is currently being used by the old dynamic space. The amount allocated could be increased some if a lot of data was kept the last time a GC was performed.

2. Scan the roots of Lisp moving any Lisp objects from old dynamic space to new dynamic space leaving GC forward pointers behind. The roots of Lisp would include the Lisp registers, control and binding stack, and static space.

3. Scavenge new dynamic space. That is, all the objects in new dynamic space that contain Lisp object must be scanned for pointers into old space. All accessible objects in old space must be transported to new space.

4. Use `vm_deallocate` to deallocate the old dynamic space, since it is no longer in use.

The above GC algorithm should work well under Mach. There only needs to be a free pointer (the first free location in new dynamic space) and an end pointer (the first location that is unusable). On a machine with 32 registers, these might as well be in registers. On other machines, leaving these in memory is probably best.

Under Unix, a similar algorithm could be used, but if new dynamic space is not contiguous because of C allocation problems, GC will have to be more complicated. It may have to allocate large objects in a different area than the current allocation area.

7 Genesis

There are two options with genesis. If the a.out file format is used for Lisp save files as it should if only Unix features are used, then Genesis will have to be modified heavily. If Mach memory allocation and file mapping features are used, the the current Genesis can be used without too much modification. Although, it would make sense to have a way for data areas used by the miscops to be allocated at miscop load time and any references to these areas be resolved by genesis rather than using fixed memory location as is currently done.

If the Unix a.out format is used, Genesis must be modified to generate an a.out executable file rather than the current Lisp core file format. I do not believe that this is difficult. To use the a.out format, the following sequence of operations need to be performed:

1. Read the C start up file object code and allocate space for it at the front of what will become the text segment. Allocate space for the C data at the start of what will be come the data segment. Note: On the IBM RT PC this is straight forward and can be done easily from the executable file created by `ld`, since text starts at #x0 and data starts at #x10000000. On other machines it will be necessary to make sure that the data segment starts at a location high enough to allow space for all the Lisp read-only data. This may be done using ld switches or by having a small assembler language routine that allocates a few megabytes of space in the text segment.

2. Read the Lisp miscop files and place them in the text segment after the C code. Note that some of the current miscops need to access fixed memory locations. A more ambitious scheme would allow the assembler to define data areas for the miscops that need data. The allocation tables needed by Lisp are an example where this is would be useful. If this scheme is adopted, these data structures would be allocated after the C data.

3. The Lisp files needed to run a kernel core should now be loaded. All code/function objects should be allocated in the text segment. All other objects should be loaded in the data segment. This is not an optimal arrangement. Certain strings (e.g., documentation strings and symbol names) and other constants should also be put in the text segment. This requires the compiler to generate information so that the cold loader knows what objects are read-only.

4. Write out an a.out file with the new text and data segments created by the above process. Also the symbol table information contained in the original C start up code for Lisp should be copied to this file so that loading of foreign functions can use this symbol table information.

8 Loading C code

Loading C code should be relatively simple. As is currently done, it will be necessary to run ld with the appropriate switches to generate an incremental load file using the lisp executable file as the base (or the most recent intermediate file created by a previous load of foreign code in the currently running Lisp). This incremental load file should be linked so that it will be placed at the end of the current lisp static area.

A problem that may occur here is that the space between the end of the current static area and the beginning of dynamic 0 space is too small for the resulting object file. In this case, it is necessary to perform a garbage collection so that more space can be allocated to static space from dynamic 0 space.

9 Purification

Purification should not be as necessary in the new system as it was in the old. With all the functions in a file contained in one combined function/code object, the function object and code vector will always be near one another. The important thing to do with purification is to make sure that all the constants pointed at by a function/code object be near it. The only things that may cause problems are symbols, because other constants should be near the function/code object since they are loaded at the same time. Also, some function/code objects should be near one another if they share a large number of constants. Initially, I don't think it is necessary to implement purification. At some time in the future purification should be combined with the save Lisp process so that the Lisp save file has somewhat better locality than it would otherwise.

In the future the purification process should be done while a Lisp core file is being saved. This will allow the system to write out either an a.out file format or the current Lisp core file format file in which everything may have been relocated to reduce the amount of paging. Purification should effectively do a complete GC of read-only, static, and dynamic space relocating everything into read-only and static space with all objects relocated so that function/code objects that call one another or share many constants are near one another. The information necessary to do this can be retrieved from the function/code objects. The number of references to an object from a function/code object could be obtained, but it probably isn't worth the effort. Using the reference information, a better ordering of all Lisp objects could be made.

10 Saving Lisp Cores

There are two options for saving a Lisp core file. One is to use the a.out format which would be the preferred method if Unix is to be supported. Under Mach, retaining the current (or similar) format has some advantages. In particular, all the memory allocation and stack problems are no longer something to worry about. The current save miscop should be rewritten in Lisp for portability reasons.

The save process under Unix should use the a.out file format and effectively do purification while creating the file. The Unix manuals contain a description of a.out file format. In the following, I am assuming that all three stacks have been combined into one and the standard Unix stack is being used for this combined stack. To create a Lisp `a.out` file, the following steps are necessary:

1. If Lisp is currently allocating space in the dynamic 0 area, a normal GC should be performed to move all the data into dynamic 1 space.

2. If there is a gap between the text and data segments as on the IBM RT PC, allocate space to hold the new read-only data area in the text area. On machines without this gap, read-only data must be placed in the static area. All references to read-only below assume this.

3. Do a purifying GC in which objects are moved into the new read-only area and to the end of the current static area. The static area will grow and either more memory must be allocated to dynamic space or the size of the dynamic areas must be adjusted appropriately after this purifying GC has finished.

4. The Lisp is now ready to be saved. First the header page must be written out. This header page includes a magic number that says this is an a.out file. It also includes the size of the text segment (read-only data), the size of the initialized data segment (static data plus amount of stack that must be saved), the size of the symbol table, and the size of the string table must be written. The size of the uninitialized data segment should be 0.

5. The text (read-only data) segment must be written. It must be rounded off to a page boundary.

6. The initialized data (static data) segment must be written. The part of the stack that Lisp needs to run must also be written to the file now. This segment also must be rounded off to a page boundary.

7. The symbol table and string table information from the original Lisp a.out file or the last intermediate a.out file created by loading C code must be copied to the file.

At this point, a new `a.out` file should have been created. This file should be made executable and it should be possible to run and have Lisp restart.

One important thing to notice is that when the saved Lisp file starts executing, it must copy the stack information out of the initialized data segment into the proper place on the Unix stack. This should not be difficult, but may require someone to write some assembly code to do this. Note that the restarted Lisp should make sure the stack is in the location it expects—otherwise the OS has changed out form underneath it and it will be necessary to recreate the saved file on the new version of the operating system.

11 Summary

I have tried to give a brief summary of how memory should be layed out under two assumptions: one using Mach virtual memory features and the other using standard Unix memory allocation. My belief at this point is that it would be nice to run in a standard Unix environment, but that it will complicate everything to such a degree that I don't think it is worth doing. When the system is running stably on a couple of machines, it might be worth spending the large amount of time necessary to come up with a good scheme for dealing with standard Unix (if you wait long enough maybe it will improve to the point that some of the serious problems are corrected).

My recommendation would be to use Mach features and do the following:

- Use the current core file format and the current Lisp start up code and map the various regions of memory Lisp uses into the process. Using the a.out file format would be convenient, but makes it difficult to have stacks out of the way.

- Use the Unix stack provided by Mach as the non-lisp or number stack. This means the compiler should specify that whatever register is used for this stack be used as the number stack register. This stack always grows downward.

  Keep the binding and control stacks separate and place them just below the Unix stack. On the IBM RT PC the Unix stack is in segment 13, the binding should be in segment 12, and the control stack in segment 11. These latter tow stacks should also grow downward.

- `Vm_allocate` the rest of whatever segment the C data is in. Load C code and user defined miscops into this area and move Lisp objects here during purification. If, as on the IBM RT PC, there are separate text and data segments, then read-only data could be moved into the text segment. This probably isn't worth worrying about.

- Dynamic space should be allocated using `vm_allocate`. When a GC occurs, new dynamic space should be `vm_allocated` at that time based on the size of the current dynamic space and how much space was retained during the previous GC. The initial amount of space allocated to dynamic space should be specified by a command line switch with a reasonable default value (such as 2 megabytes). `Vm_deallocate` the old dynamic space when the GC is finished.