CLISP's memory management

Bruno Haible
Mathematisches Institut II
Universität Karlsruhe
Kaiserstraße 12
D – 76131 Karlsruhe
Email: <haible@ma2s2.mathematik.uni-karlsruhe.de>

19 February 1995

Categories: Garbage collection, OS support for MM

1 General overview

CLISP is a Common Lisp implementation for general-purpose hardware, with freely available source. It has an interpreter and a bytecode compiler.

The GC is compacting. To ensure that all GC roots are traversible at GC time, a second stack, for Lisp objects only, is employed.

CLISP is designed to be portable to any kind of OS – from diskette-only home computers, 1 MB RAM, to workstations with 64 MB RAM. It is prepared to operating systems which provide malloc only, but can make use of mmap, shmget, mprotect system calls on systems where they are available.

The computation time spent on GC is typically around 10% of the total computation time.

2 The GC algorithms

The design goal for GC was to make best possible use of the available memory and use secondary storage as few as possible. This also implies virtual memory efficiency on VM systems.

As a consequence:

- No mark & copy collector is used, since this would require twice as much RAM as the data set actually needs.

- A compacting collector is used, in order to be able to give back to the operating system memory pages which have only a few objects left.

- Generational GC is used, to reduce the amount of memory accesses during a GC.

Memory is organized into objects, some of them containing pointers. The object's type (partially in the pointer itself, partially on the heap) describes the location of pointers within the object. There are two kinds of heaps: one for fixed sized objects (two pointers each) – call them "type F objects" –, one for objects of other sizes – call them of "type O". Since the latter is traversed from left to right, it is necessary to store the object's length at the beginning of the object.

The classical compaction algorithm is five-pass:

- Pass 1: Mark the live objects. To avoid stack overflow, this is implemented in a pseudo-recursive manner, using the pointer reversal trick. There is one single marking subroutine, with the type dispatch and all special cases for the individual types being inline. This makes this routine pretty fast (no function call overhead).

- Pass 2: For every type O object, compute the address it will be moved to, and install a forward pointer from the current to the future location.

- Pass 3: Walk through all objects, replacing pointers to current locations by pointers to new locations.

- Pass 4: Compact the type F objects using the Morris algorithm. This requires two linear passes, in opposite directions.

- Pass 5: Compact the type O objects.

The compactions in pass 4 and 5 remove the holes between the objects by "sliding" them down. As a consequence, objects are always kept in the order of their creation, thus retaining the amount of locality present in the data.

This description holds for systems on which two independently growing heaps can be implemented, either via `mmap()` or `shmat()`.

On other systems, or when memory is rare and should not be abused, CLISP allocates memory in smaller chunks (pages, around 4 KB). Extra large objects are accomodated in extra pages. The above algorithm still applies, *mutatis mutandis*. The allocation of type O objects of size n, however, requires the quick search for a page that contains at least n free bytes. This search is achieved through the use of a balanced binary tree which keeps the type O pages sorted according to their size.

Memory requirements for these GC algorithms:

- One bit per pointer has to be reserved for GC purposes, used as marking bit.

- One pointer per type O object, used as a forward pointer.

3 Generational GC algorithm

In order to reduce the amount of memory touched during a single GC, a generational GC is employed, as described in [Wilson].

Basically, the heap is divided into an "old generation" and a "new generation". "Full" GCs clean up both, while "partial" GCs clean up the new generation only. After a full GC, the new generation is empty.

During a partial GC, the pointers in the old generation is considered part of the root set, and pointers into the old generation need not be adjusted because objects in the old generation don't move.

To every page in the old generation, we keep a cache containing all pointers into the new generation. This way, walking through this page (as part of the root set) is equivalent to walking through the cache. This is done if the cache's size is less than 1/2 the page size. If the cache would be larger than this threshold, don't waste space for the cache, don't cache the page at all.

Thus, most of the old generation pages are not touched by a partial GC. The cached pointers are updated by the GC (they point into the new generation), but the actual pointers in the page are not – since this would mean an unnecessary page fault if the page is currently on secondary storage. Instead, the page is marked neither readable nor writeable, and the pointer update is delayed until the next access to that page – which would force the page into RAM anyway.

Old generation pages which have a valid cache are marked readable but not writeable, in order to notice the next write access to that page and mark the cache invalid at that moment.

The policy when to do a partial and when to do a full GC is very simple in CLISP: If at the end of a GC the new generation's size is > 25% of the old generation's size, the next GC will be a full one.

Requirements for the generational GC algorithm:

- The operating system supports `mmap()` for anonymous memory, or `shmat()`, and `mprotect()`. CLISP's generational GC works on Linux, SunOS, NeXTstep, OSF/1, IRIX.

- Within every object, the pointers form a contiguous block. (This is a technical restriction only.)

4 What the OS could do better

The GC algorithms could probably do better if it were possible to access the dirty bits of memory pages in a portable way.

The following have caused portability problems:

- Syntax of `mmap()` flags. Some use `/dev/mem`, some use `MAP_ANONYMOUS`. Some need `MAP_FILE`, AIX needs `MAP_VARIABLE`.

- `shmat()` is crippled or doesn't work reliably on some systems. (Bad interactions with swapping etc.)

- System calls like `read()`, `write()` return error `EFAULT` if they encounter a write or read protected page. They should call the `SIGSEGV` signal handler instead.

- How to get at the fault address when a `SIGSEGV` signal handler gets called, is highly system dependent. It could well afford a bit of standardization.

Shared memory efficiency suffers from the fact that there is no equivalent of `munmap()` for shared memory which would tell the OS that a range of pages can be thrown away.