GCHAX

GCHax contains functions that are useful for tracking down storage leaks, i.e., objects that should be garbage but do not get garbage collected. There are functions for examining reference counts, locating pointers to objects, and finding circularities (which are among the chief culprits in storage leaks).

Typically, you might turn to GCHax when you notice that `STORAGE` claims there are more instances of a data type in use than you believe there should be.

Installation

Load `GCHAX.LCOM` from the library.

Functions

Storage

The function `STORAGE` displays statistics on the amounts and distribution of the virtual memory space that has been allocated. If you suspect your program may have storage leaks (e.g., because `(VMEMSIZE)` keeps growing without obvious reason), this function is the place to start to get an indication of which kinds of objects are not being reclaimed. `STORAGE` is part of the standard Lisp sysout; you need not have loaded GCHAX to use it.

`(STORAGE` *TYPES PAGE-THRESHOLD IN-USE-THRESHOLD*`)` [Function]

With no arguments, `STORAGE` displays statistics for all data types, along with some summary information about the space remaining. The optional arguments let you refine the display.

If *TYPES* is given, `STORAGE` only lists statistics for those types. *TYPES* should be a type name or list of type names.

If *PAGE-THRESHOLD* is given, then `STORAGE` omits types that have fewer than *PAGE-THRESHOLD* pages allocated to them. The default *PAGE-THRESHOLD* is `2`, so types that are not currently in use (consume no storage) do not appear unless you specify a *PAGE-THRESHOLD* of zero.

If *IN-USE-THRESHOLD* is given, then `STORAGE` omits types that have fewer than *IN-USE-THRESHOLD* instances in use (allocated and not yet freed).

For example, `(STORAGE '(ARRAYP BITMAP))` lists only statistics for the types `ARRAYP` and `BITMAP`; `(STORAGE NIL 6)` lists only statistics for data types that have at least six pages allocated. `(STORAGE NIL NIL 100)` lists only statistics for data types that have at least 100 instances still in use.

The `STORAGE` function displays, for each Lisp data type, the amount of space allocated to the data type, and how much is currently in use. The display looks something like this:

```
Type            Assigned        Free items     In use   Total alloc
                pages [items]
FIXP               66    8448       7115         1333        447038
```

```
FLOATP            24    3072       2412          660         734877
LISTP           2574 ~298584       5294      ~293290        3545071
ARRAYP             8     512        245          267          48199
    . . .
```

`Type`	Is the name of the data type, as given to `DATATYPE` or the Common Lisp `DEFSTRUCT`.
`Assigned`	Is how much of your virtual memory is set aside for items of this type. Memory is allocated in quanta of two pages (1024 bytes). The numbers under Assigned show the number of pages and the total number of items that fit on those pages. The tilde (~) on the `LISTP` line indicates that the number is approximate, since cdr-coding makes the precise counting of lists impossible—the amount of memory consumed by any particular list cell varies depending on its contents and how it was allocated.
`Free items`	Shows how many of the assigned items are available to be allocated (by the Interlisp create or the Common Lisp make- constructs); these constitute the free list for that data type.
`In Use`	Shows how many items of this type are currently in use, i.e., have pointers to them and hence have not been garbage collected. If this number is higher than your program seems to warrant, you may want to look for storage leaks. The sum of Free items and In Use is always the same as the total Assigned items.
`Total Alloc`	Is the total number of items of this type that you have ever allocated (created), or at least since the last call to `BOXCOUNT` that reset the counter.

`STORAGE` also prints some summary information about how much space is allocated and available collectively for fixed-length items (mainly data types, both user and built-in), variable-length items (arrays, bitmaps, strings), and symbols. The variable-length items have fixed-length headers, which is why they also appear in the printout of fixed-length items. For example, the line printed for the data type `BITMAP` says how many bit maps have been allocated, but the figure displayed as "assigned pages" counts only the headers, not the space used by the variable-length part of the bitmap. The variable length portion is accounted in the summary statistics for "ArrayBlocks", where it is lumped with all other users of variable-length space, as it is not possible for the system to more finely discriminate the users of the space.

```
Data Spaces Summary
                              Allocated           Remaining
                                Pages               Pages
Datatypes (incl. LISTP etc.)      4370            \
ArrayBlocks (variable)            5770            -- 47758
ArrayBlocks (chunked)             2626            /
Symbols                           1000                1048

variable-datum free list:
le 4            18 items;       72 cells.
le 16           84 items;      865 cells.
le 64           38 items;     1019 cells.
le 256          76 items;     7580 cells.
le 1024          2 items;     1548 cells.
le 4096         11 items;    18568 cells.
le 16384         1 items;     4864 cells.
others           2 items;    59565 cells.
```

```
  Total cells free:    94081  total pages:  736
```

In the summary, Remaining Pages indicates how many more pages are available to be allocated to each type of datum. There is a single figure for both fixed- and variable-length objects, because they are allocated out of the same pool of storage.

Variable-length objects are allocated in two different ways, reflected in the items "variable" and "chunked." The distribution of the former among several different sized free lists is shown next.

Storage Leak Tracking Functions

The functions in GCHax are oriented toward finding leaks that involve items of some data type not getting garbage collected.

There are two main kinds of leaks:

- Items that are unintentionally being held onto
- Items that no user structure is pointing to but are not collected because of the nature of the garbage collector

Examples of the former are structures assigned to global variables and left there after the program finishes.

Examples of the latter are principally circular structures — structures where you can follow a chain of pointers from an object that eventually returns to the same object. Circular lists, such as you get from `(NCONC A A)`, are a special case of circular structures. See comments in the Limitations section below.

Note: All functions listed below have names beginning with \ to remind you that you are dealing with system internals, and to proceed with at least a little caution. Although these functions are generally safe, in that their casual use will not cause arbitrary damage, you certainly can produce unintended side effects.

In particular, the functions `\SHOWGC` and `\COLLECTINUSE` have modes in which they return a list of some kind of pointer; beware of unintentionally holding on to such a list (e.g., by having it get onto the history list), thereby preventing the eventual garbage collection of any of those pointers.

Useful for keeping values off the history list are the Executive command SHH for completely inhibiting history list entry, and the idiom (PROG1 NIL operation), e.g., `(PROG1 NIL (INSPECT value))` to inspect a structure without holding on to a pointer to the inspect window. You may find it convenient to define your own Exec command to do inspection, e.g.,

```
(DEFCOMMAND IN (OBJ TYPE)
(PROG1 NIL (INSPECT OBJ TYPE)))
```

The reference counts of all objects in the system are maintained in a global hash table, called the GC reference count table. Some or all of its contents can be viewed with the following function:

`(\SHOWGC` *`ONLYTYPES COLLECT FILE CARLVL CDRLVL MINCNT`*`)` [Function]

Displays on *FILE* (default `T`) all objects in the GC reference count table whose reference count is at least *MINCNT*, whose default value is `2`.

If *ONLYTYPES* is given, it is a list of data type names to which `\SHOWGC` confines itself.

If *COLLECT* is `T`, `\SHOWGC` returns a list of all the objects it displays.

CARLVL and *CDRLVL* are print levels affecting the displaying of lists; they default to two and six, respectively. In the listing, collision entries in the reference count table are tagged with a *. Reference count operations on pointers in collision entries are much slower than on noncollision entries.

Objects with reference count of one (1) do not appear explicitly in the reference count table, so cannot be viewed with `\SHOWGC`, even if you set *MINCNT* as low as 1.

Note that if *COLLECT* is `T`, then the reference count of all the collected items is now one greater, due to the pointer to each from the returned list.

`(\REFCNT` *`PTR`*`)` [Function]

Returns the current reference count of *PTR*. Pointers that are not reference counted (e.g., symbols and small integers) are considered to have reference count 1. Since pointers from the stack (e.g., `PROG` variables) do not affect reference counts, it is possible for the reference count of an object to be zero without the object being garbage collected.

Note: If you call `\REFCNT` from the Common Lisp interpreter, e.g., by typing it at top-level, the answer is almost always too large by 1, as the interpreter itself holds reference-counted pointers to the arguments to the function it is calling. The same problem besets `\FINDPOINTER` (below). The problem does not exist from the Old Interlisp Exec, which uses the Interlisp interpreter. You can also avoid the problem by explicitly invoking the Interlisp interpreter; e.g.,

```
(EVAL '(\REFCNT expression)).
```

`(\#COLLISIONS)` [Function]

Returns a list of four elements:

- Number of entries in the reference-count table, i.e., the number of objects in memory whose reference count is not 1
- Number of entries that are in collision chains
- Ratio of these numbers, i.e., the fraction of all entries that are in collision chains
- Ratio of the number of entries to the size of the hash table

`(\#OVERFLOWS)` [Function]

Returns a list of four elements like `\#COLLISIONS`, but instead counts only objects whose reference count has overflowed (is greater than 62). Reference count operations on such objects are significantly slower than on other objects.

`(\COLLECTINUSE` *TYPE PRED*`)` [Function]

Is useful when (`STORAGE` *TYPE*) shows more objects in use than you think is right, but you can't find any such pointers yourself.

TYPE is a data type name or number other than `LISTP`. `\COLLECTINUSE` returns a list of all objects of that type that are thought to be in use, i.e., not free.

If *PRED* is supplied, it is a function of one argument. `\COLLECTINUSE` returns only objects for which *PRED* returns true. *PRED* must not allocate storage; you probably want it to be a compiled function.

Note: `\COLLECTINUSE` should be used with care. In a correctly functioning system, `\COLLECTINUSE` is generally safe. However, if the free list of *TYPE* has been smashed so that some free objects are not on it, this function can make matters much more confused, especially if the first 32-bit field of the data type in question contains a pointer field.

(`\FINDPOINTER` *PTR COLLECT/INSPECT? ALLFLG MARGIN ALLBACKFLG*) [Function]

Provides a brute-force approach to answering the question, "Who has a pointer to *x*?" `\FINDPOINTER` searches virtual memory, looking for places where *PTR* is stored. The search is not completely blind: unless *ALLFLG* is true, it does not look in places that cannot have reference-counted pointers, such as pname space or the stack. However, if the reference count of the object is zero, `\FINDPOINTER` searches the stack (and only the stack, if *ALLFLG* is `NIL`), since in this case there is no hope of finding pointers in the usual reference-counted spaces. If *ALLFLG* = `:STACK`, then `\FINDPOINTER` searches the stack in addition to places that contain reference-counted pointers, but not other unlikely places.

`\FINDPOINTER` prints out a description of each place *PTR* is found. If it is found in a list, it asks whether to recursively search for pointers to the list, so you can track lists back to a more identifying place, such as a symbol value cell or some data type. It recurses without asking if *ALLBACKFLG* is true. If *PTR* is found in a typed object, `\FINDPOINTER` names the field, if the data type declaration is available, and asks if you want to recursively search for pointers to this object. In either case, the search stops once enough places have been found to account for *PTR*'s reference count (unless *ALLFLG* is `T`).

If *COLLECT/INSPECT?* is true, `\FINDPOINTER` saves the identifiable pointers in a list. If *COLLECT/INSPECT?* = `COLLECT`, the list of pointers is returned as value; otherwise, it is offered for inspection.

MARGIN is the left margin (in units of characters) by which the reports of locations are initially indented. The default is zero. Recursive searches for pointers are indented relative to this position.

The current version does not know how to parse array space, so if *PTR* is found in an array, the best it can do is print the memory address where it found it, usually something of the form {}#nn,nnnnn. In addition, `\FINDPOINTER` doesn't even try to find *PTR* as a literal inside a compiled code object, since such references are not cell-aligned. Thus, `\FINDPOINTER` is really most helpful if the pointer is stored in fixed-length data space (e.g., in a field of a data type, or as the top-level value of a symbol); fortunately, this handles most of the interesting cases in practice.

Note: Of course, since it touches (potentially) a huge percentage of your virtual memory, `\FINDPOINTER` is completely disruptive of your working set.

(`\FINDPOINTERS.OF.TYPE` *TYPE FILTER*) [Function]

Calls `\FINDPOINTER` on each pointer in use of type *TYPE* that satisfies *FILTER*, a function of one argument, the pointer. A *FILTER* of `NIL` is considered the true predicate. *FILTER* can also be a list form to evaluate in which the variable `PTR` is used to refer to the pointer in question.

`\FINDPOINTERS.OF.TYPE` is essentially the same as

```
(for PTR in (\COLLECTINUSE TYPE)
   when <FILTER is satisfied>
   do (\FINDPOINTER PTR))
```

except that it takes care to discard the cells of the list returned from \COLLECTINUSE before calling \FINDPOINTER, to avoid seeing one extra reference per object.

For example,

```
(\FINDPOINTERS.OF.TYPE 'STREAM '(NOT (OPENP PTR)))
```

searches for pointers to all streams that are not currently open.

`(\SHOW.CLOSED.WINDOWS)` [Function]

Collects all windows that are not currently open or icons of open windows, then opens each window one by one.

For each window, you are prompted to press the left mouse button to close the window and go on to the next, or press right to do something different. In the latter case, you are prompted again to press the left button if you would like to search for pointers to the window, using `\FINDPOINTER`, or press the right button to just leave the window open on the screen and proceed.

Returns the total number of windows examined.

`(\SHOWCIRCULARITY` *OBJECT MAXLEVEL*`)` [Function]

Follows pointers from *OBJECT*. If it finds a path back to itself, it prints that path. This function is not exceptionally fast, and deliberately (for performance reasons) does not detect circularities in lists; it simply bottoms out on lists at *MAXLEVEL*, which defaults to `1,000`. Circular lists are usually obvious enough anyway.

`(\MAPGC` *MAPFN INCLUDEZEROCNT*`)` [Function]

Maps over all entries in the GC reference count table, applying *MAPFN* to three arguments: the pointer, its reference count (an integer), and *COLLISIONP*, a flag that is `T` if the entry is a collision entry. Entries with reference count zero are not included unless *INCLUDEZEROCNT* is `T`. This function underlies `\SHOWGC`. Some care is required in the writing of *MAPFN*; it should try to minimize any reference-counting activity of its own, and in particular avoid anything that would decrement the reference count of the pointer passed to it.

Limitations

GCHax is not very useful for finding ordinary circular lists, as the typical system has vast amounts of list structure, with nothing to distinguish the interesting ones.

However, if the circular list also contains instances of user data types, then those data types will tend to show up as overallocated, and hence amenable to the search functions in this module.

`\FINDPOINTER` does not know how to locate pointer arrays of more than 64 elements, so it is not helpful if a pointer you seek is located only in such an array.

[This page intentionally left blank]