CSL Notebook Entry

To	CedarUsers	Date	October 29, 1982
From	Paul Rovner	Location	PARC/ CSL
Subject	Concise Guide to Storage Management in Cedar		

File [Indigo]<Cedar>Documentation>SafeStoragePrimer.tioga, .press

XEROX

Release as [Indigo]<Cedar>Documentation>SafeStoragePrimer.tioga, .press
Draft [Indigo]<PreCedar>Documentation>SafeStoragePrimer.tioga, .press
Last edited by Paul Rovner, October 26, 1982 7:51 pm

Abstract SafeStorage.df comprises the parts of Cedar that provide runtime support for Cedar language features: storage allocation, garbage collection, REF ANY (NARROW, WITH ... SELECT) and ATOMs. This memo is a concise guide to the most commonly used features of SafeStorage.df for the beyond-novice client who understands the Cedar language. More detailed and complete documentation can be found in [Indigo]<Cedar>Documentation>SafeStorage.tioga (and .press).

Overview of the most commonly used public interfaces of SafeStorage.df

SafeStorage.mesa provides facilities for managing safe (i.e. collectible, REF) storage and for controlling the collector. It is by far the most commonly used interface purveyed by SafeStorage.df (hence the choice of .df file name).

UnsafeStorage.mesa provides facilities for managing unsafe (i.e. non-collectible, LONG POINTER) storage.

RTTypesBasic.mesa provides facilities for dealing directly with the basic runtime type system (via RTBasic.Type's), e.g. to specify finalization for a Type or get the Type of the object addressed by a REF.

When and when not to create your own ZONE

Generally, a poor match between the use of a ZONE and the design of its allocation strategy will lead to inefficient operation. If you are concerned with such performance issues, you will want to select an allocation strategy that fits your usage. (It may also help for us wizards to explore other allocation strategies, perhaps as additional options. It's on the list.)

In current Cedar, the system ZONE (i.e. the one you get if you use NEW rather than z.NEW) has a single free list, and hence is susceptible to fragmentation. Heavy use for objects of widely different sizes affects overall system performance adversely.

If your package or application explicitly allocates (via NEW) more than about 2K words of storage, it is likely that the world would be better if you created at least one special ZONE for exclusive use by your package.

Creating your own ZONE and guidelines for selecting ZONE parameters

```
SafeStorage.NewZone: PROC
        [ sr: SizeRepresentation _ prefixed,
          base: Base _ nullBase, -- default will use the RootBase
          initialSize: LONG CARDINAL _ 0 -- words
        ] RETURNS[ZONE];
```

Zones acquire quanta as they are needed. Occasionally, this causes a new Space to be created in VM. The only limit on the number and run-length of quanta that can be acquired by a zone is the amount of available VM. Quanta acquired by a zone remain assigned to it until the zone is "trimmed" (see SafeStorage.TrimZone). All zones get trimmed automatically when the Cedar allocator discovers that VM is getting tight. This (discovery) happens very rarely in current Cedar. A package would do well to invoke TrimZone explicitly when there is a lot of free storage in the zone, e.g. at the end of a big computation.

Of the three parameters to NewZone, the choice of sr: SizeRepresentation affects performance the most. You can safely ignore the others.

A quantized zone is good if there are many objects of the same size and type, or of a few (roughly a dozen or less, say) different sizes and types. Sequences and variant records are examples of types that can have different sized objects. An integral number of quanta (each 1K words, 4 pages) is assigned to each (size, Type) combination serviced by a quantized zone. This means that use of a quantized zone for allocating just a few words of a particular (size, Type) is wasteful of VM.

Quantized zones are good for small objects because there is no per-object storage overhead. The maximum size of an object from a quantized zone is one quantum. The minimum size is 2 words.

Prefixed zones are more general (e.g. for TEXT objects), but their allocator is slower, each object requires extra storage cells to carry size and type information, and prefixed zones are susceptible to fragmentation. The maximum size of an object from a prefixed zone is (LAST[CARDINAL] - 1) words. The minimum size is 4 words. The storage overhead per prefixed object is 2 words. Adjacent blocks on the free lists of all prefixed zones are merged at the end of every collection.

All object lengths are even (rounded up if necessary). All objects begin on even word boundaries.

ZONEs are collectible objects. A ZONE will be reclaimed automatically if no REFs to it remain accessible and if no REFs remain accessible to objects in any of its quanta.

Facilities in Cedar for creating UNCOUNTED ZONEs

```
UnsafeStorage.NewUZone: PROC
        [ sr: SizeRepresentation _ prefixed,
          initialSize: LONG CARDINAL _ 0 -- words
          typeRepresentation: BOOL _ FALSE
        ] RETURNS[UNCOUNTED ZONE];
```

This guy creates a new UNCOUNTED ZONE. NewUZone should be used instead of Heap.Create (it allows more than 64K per UNCOUNTED ZONE and allocation is more efficient). The same allocator used for collectible objects is used here. Appropriate cleverness is employed to do none of the work associated with reference counting. Cedar allocation microcode is used (!).

The comments above about quanta and SizeRepresentation for ZONEs apply to UNCOUNTED ZONEs, except that no automatic trimming is done (see UnsafeStorage.TrimUZone and UnsafeStorage.MergeUZone) and UNCOUNTED ZONEs are not collectible objects.

Basic Dealings with Runtime Types; Finalization

Each Cedar object carries a runtime representation of its TYPE (i.e. the argument to the NEW that created it). This is an RTBasic.Type, which is a 14-bit index into a table that is maintained by the runtime system. Information therein is used to support NARROW, WITH <ref any> SELECT FROM, and runtime access to symbolic information about TYPEs. The special language construct CODE[<TYPE expression>] evaluates to a CARDINAL that is the 14-bit index for the specified TYPE.

RTTypesBasic.Mesa provides facilities for dealing directly with the basic runtime type system (via RTBasic.Type's), e.g. to get the Type of the object addressed by a REF, to determine whether two Types are equivalent, or to specify finalization for a Type.

BEWARE: it is wrong to expect that if two Types are not equal, they do not represent the same or equivalent TYPEs. Use RTTypesBasic.EquivalentTypes. Every Type has a unique "canonical Type". Two canonical types are equal IFF the original TYPEs are equivalent .

It is possible for a client to specify actions to take when an object of a particular Type is no longer accessible to any client of the package that created it. Such *package finalization* actions might include (for example) removal of a reference to the object from a package-maintained cache. See SafeStorage.press for the details.

Caveats about the use of REFs: limitations and unsafe operations

Garbage collection in Cedar is based on the maintenance of reference counts. The inherent problem with this scheme is that the collector will miss circular structures that are reclaimable. Clients should avoid the use of circular structures when possible or break circularities by storing NIL when possible.

Various clever tricks are used to minimize the cost of maintaining reference counts, but (as usual) these have limitations. The major one is that reference counts get pinned when they reach a maximum (77B in the soon-to-be-current system). Clients should avoid building data structures that have large numbers (~thousands) of pinned reference counts when possible.

Cedar does provide a "TraceAndSweep" collector that will reclaim circular garbage and inaccessible objects that once had pinned reference counts, but it wedges the world for half a minute on a Dorado to do its work. The TraceAndSweep collector will be invoked automatically when the Cedar allocator finds itself in dire circumstances. This happens very rarely.

SafeStorage.ReclaimCollectibleObjects provides a way to invoke the TraceAndSweep collector explicitly.

Another limitation derives from the use of a separate data structure for maintaining reference counts that are greater than one (the "reference count table"). Clients should avoid building data structures that have large numbers (~thousands) of such objects.

Come see me if you find yourself tempted to use LOOPHOLE in any way relating to REFs.

Client-alterable system parameters: controlling the garbage collector

SafeStorage.mesa provides access to parameters and operations that control the garbage collector.

The garbage collector gets initiated automatically whenever "CollectionInterval" collectible words have been allocated since the last collection was initiated. The collector runs in the background, concurrent with client allocation and reference-counting activity. Per-process accounting of storage utilization is done; if a process would trigger a collection while the collector is running AND that process has allocated more than "CollectionInterval" words since the collector was last triggered then the process will be suspended until the collector finishes.

It is possible to change the "CollectionInterval" system parameter. This balances the cost of collection against the unreclaimed storage "high water mark".

Garbage collection will also be invoked automatically by the reference-counting machinery when its tables become full or when more quanta than are available are required from VM. In the latter case, all ZONEs will be trimmed and free Spaces will be returned to Pilot after the collection finishes. It is possible to change the threshold on Cedar allocator VM that triggers such a collection. This balances the cost of trimming all zones and freeing Spaces against the allocator’s VM high water mark.

If the reference count table overflows during a collection then reference counting and the incremental collector will be disabled. The trace-and-sweep collector will be invoked at this time to perform a more leisurely and thorough collection (recovering circular structures and inaccessible objects with pinned reference counts) and reconstruct the reference count table.

SafeStorage.ReclaimCollectibleObjects is the client interface for explicit invocation of the collector.