Garbage collection - Part 3: Grievances

2009-03-24 / rev 1 / lhansen@adobe.com

API

GC.h and GC.cpp files together comprise 5000 lines of code, one feels there's room for more modularity. In particular, GC.h mixes internal structures with the external API. (Hard to avoid in C++ but there's room for improvement.)

MMGC_GET_STACK_EXTENTS is an architecture and compiler specific porting interface, it surely does not belong in this file.

The class GCWorkItem is internal to the GC and need not be exposed here.

The ZCT is not used outside MMmgc at all and need not be exposed, and could be separated from this file.

Hidden porting API: GC::CleanStack implements a compiler-specific (at least) mechanism for getting the stack top. Should be factored.

Sundry remarks, gripes, etc

RCPtr

RCPtr is a fat pointer that keeps alive the RCObject that it holds: it increments the reference count on initialization or assignment, and decrements it on destruction or overwriting. It tracks RC pointers from unmanaged memory.

Summary of conversation with Tommy to clear up something mysterious:

```
Lars Hansen
In RCPtr:
inline bool valid() { return (uintptr_t)t > 1; }
Why "> 1"?
t is the object that's held, not a counter field.
normally it should be a pointer, that's the way it's used throughout

Tommy
the name table in AvmCore is DRC pointers
1 == a DELETED key in the hashtable
ditto for Namespace intern table
```

ZeroPtr

The class ZeroPtr is used as a fat pointer that is initialized to NULL (or a derived pointer) on creation and reset to NULL on destruction, presumably to avoid false references from the stack.

Issue: ZeroPtr is not used by the AVM (except in a couple of APIs that we don't use) or in the Player. There is a comment in the Player to the effect that ZeroPtr isn't viable but may be useful for debugging:

```
 the ZeroPtr stuff has been determined to hurt performance too much so we don't
 do it, in case we think we have a situation where a stack pointer is causing a
 GC object to stick around that shouldn't turning on ZeroPtr should help make
 that determination
#if 1
typedef ScriptObject* ScriptObjectp;
typedef FunctionScriptObject* FunctionScriptObjectp;
typedef ThreadScriptObject* ThreadScriptObjectp;
#else
 auto zero'ing pointer for DRC stack cleansing purposes
typedef MMgc::ZeroPtr<ScriptObject*> ScriptObjectp;
typedef MMgc::ZeroPtr<FunctionScriptObject*> FunctionScriptObjectp;
typedef MMgc::ZeroPtr<ThreadScriptObject*> ThreadScriptObjectp;
#endif
```

Action: *Remove the class, or at least make its support depend on a feature definition. And document it (including the performance problems).*

GCHiddenPtr

GCHiddenPtr is not documented anywhere, not even on MDC, but it is used throughout the Player code. It looks like it could be a mechanism for avoiding circular references in RCObject hierarchies. Certainly it's somewhat dangerous, and it's completely incompatible with a mostly-moving collector (which we don't have but might want to have in the future).

Tommy says he has a memory that the mechanism predates the weak pointer support currently in the GC and that uses should probably be replaced by weak pointer uses.

(HIDDENPTRMASK is presumably part of its implementation not its API).

Cleaner

Utility class that does not belong in GC.h.

emptyWeakRef

emptyWeakRef and emptyWeakRefRoot are properly client matters -- they're not used by the GC, and they can be created using existing APIs. They are only used by the traits code in the VM. So move this noise out of the GC.

A dirty baker's dozen

A grab bag of annoyances that seem to niggling to report elsewhere, I should perhaps move these to bugzilla.

- *Comment rot in GCAlloc.h, referring to Windows XP, non-existent GCPageAlloc class.*
- *What is the point of* GetGCContextVariable *and* SetGCContextVariable*? Since the variables thus controlled must be named by constants in the GC class anyway, there's no difference between these and custom accessors/updaters for each of the variables. These entities are not scanned by the GC or anything.*
- *For GC::Alloc and Calloc a comment reads, "Do not call this from a finalizer." Why not? That seems like a big deal, easy to go wrong. It's not checked by means of asserts in the code.*
- *Why is AllocAlreadyLocked public? GCWeakRef calls it but only for MMGC_THREADSAFE, and GCWeakRef could be made a friend of GC, like other classes already are. GC is a friend of GCWeakRef...*
- *Note for Free() says it will "throw an assert if called during the Sweep phase", unclear what that means. Seems brittle in any case; the behavior is presumably useful and should be supported by definite semantics (which might be: ignore the request, or queue it). Note, the comment above the variable "collecting" contradicts it, it says Free just doesn't do anything during sweeping.*
- *Methods like SetQueued, ClearFinalized, SetFinalize, IsFinalized are public but undocumented.*
- *Predicates like IsFinalized and HasWeakRef return 'int' not 'bool'*
- *A comment to* IncrementalMark *reads "Do as much marking as possible in time milliseconds" but* time *is not a parameter to that method.*
- GetPerformanceCounter *and* GetPerformanceFrequency *must be documented though presumably they are somewhat intuitive (ticks and ticks per second it appears from later code). It appears that it is a requirement that we have a high-resolution timer available for incremental GC. This is a hidden porting interface, possibly very dodgy.*
- *There is a completely redundant definition of kPageUsableSpace in GC.h, nobody uses this (FixedMalloc has its own).*
- *RemoveFromZCT is exposed so that it can be called once from the Player code in one particular backward compatibility mode, but this just seems like a confused API - presumably there's something that one wants to happen that RemoveFromZCT does, but it would seem a lot cleaner to expose an API for whatever purpose it is.*
- *Collector uses tricolor invariant in documentation but never explains what the three colors mean, specifically. (The MDC page does explain it though.)*
- *We have GC::ConservativeMarkRegion declared but neither defined nor called. Ditto GC::ConservativeWriteBarrierNoSubstitute.*
- *GCAlloc::ConservativeGetMark and GCLargeAlloc::ConservativeGetMark are always defined but only used in debug code.*
- *In the department of dodgy APIs, why oh why does* Size *call* GCLargeAlloc::GetBlockHeader *instead of something more generic? Does the functionality just happen to be present in GCLargeAlloc? It seems like fairly arbitrary design.... until we find the following code in GC.cpp, explaining that the two structures are actually laid out the same.*

```
 keep GC::Size honest
GCAssert(offsetof(GCLargeAlloc::LargeBlock, usableSize) == offsetof(GCAlloc::GCBlock, size));
```

That being the case, it would seem more natural to lay out the common parts in a structure that is the base type for the block header for both large and small blocks.

- GCAlloc APIs use 'int' for itemSize, sizeClassIndex.
- Method *GCAlloc::ConservativeGetMark* returns int but takes a bool argument that is converted to int (*bool bogusPointerReturnValue*), this makes scant sense.
- A curious mixture of efficiency hacks and failure to implement same in GCAlloc; for example, FreeItem is not in-lined but it is a tiny function and is on the critical path for free objects in *SweepGuts*.