This is under construction

Conservative GC Algorithmic Overview

This is a description of the algorithms and data structures used in our conservative garbage collector. I expect the level of detail to increase with time. For a survey of GC algorithms, see for example Paul Wilson's excellent paper. For an overview of the collector interface, see here.

This description is targeted primarily at someone trying to understand the source code. It specifically refers to variable and function names. It may also be useful for understanding the algorithms at a higher level.

The description here assumes that the collector is used in default mode. In particular, we assume that it used as a garbage collector, and not just a leak detector. We initially assume that it is used in stop-the-world, non-incremental mode, though the presence of the incremental collector will be apparent in the design. We assume the default finalization model, but the code affected by that is very localized.

Introduction

The garbage collector uses a modified mark-sweep algorithm. Conceptually it operates roughly in four phases:

1. *Preparation* Clear all mark bits, indicating that all objects are potentially unreachable.
2. *Mark phase* Marks all objects that can be reachable via chains of pointers from variables. Normally the collector has no real information about the location of pointer variables in the heap, so it views all static data areas, stacks and registers as potentially containing containing pointers. Any bit patterns that represent addresses inside heap objects managed by the collector are viewed as pointers. Unless the client program has made heap object layout information available to the collector, any heap objects found to be reachable from variables are again scanned similarly.
3. *Sweep phase* Scans the heap for inaccessible, and hence unmarked, objects, and returns them to an appropriate free list for reuse. This is not really a separate phase; even in non incremental mode this is operation is usually performed on demand during an allocation that discovers an empty free list. Thus the sweep phase is very unlikely to touch a page that would not have been touched shortly thereafter anyway.
4. *Finalization phase* Unreachable objects which had been registered for finalization are enqueued for finalization outside the collector.

The remaining sections describe the memory allocation data structures, and then the last 3 collection phases in more detail. We conclude by outlining some of the additional features implemented in the collector.

Allocation

The collector includes its own memory allocator. The allocator obtains memory from the system in a platform-dependent way. Under UNIX, it uses either `malloc`, `sbrk`, or `mmap`.

Most static data used by the allocator, as well as that needed by the rest of the garbage collector is stored inside the `_GC_arrays` structure. This allows the garbage collector to easily ignore the collectors own data structures when it searches for root pointers. Other allocator and collector internal data structures are allocated dynamically with `GC_scratch_alloc`. `GC_scratch_alloc` does not allow for deallocation, and is therefore used only for permanent data structures.

The allocator allocates objects of different *kinds*. Different kinds are handled somewhat differently by certain parts of the garbage collector. Certain kinds are scanned for pointers, others are not. Some may have per-object type descriptors that determine pointer locations. Or a specific kind may correspond to one specific object layout. Two built-in kinds are uncollectable. One (`STUBBORN`) is immutable without special precautions. In spite of that, it is very likely that most applications currently use at most two kinds: `NORMAL` and `PTRFREE` objects.

The collector uses a two level allocator. A large block is defined to be one larger than half of `HBLKSIZE`, which is a power of 2, typically on the order of the page size.

Large block sizes are rounded up to the next multiple of `HBLKSIZE` and then allocated by `GC_allochblk`. This uses roughly what Paul Wilson has termed a "next fit" algorithm, i.e. first-fit with a rotating pointer. The implementation does check for a better fitting immediately adjacent block, which gives it somewhat better fragmentation characteristics. I’m now convinced it should use a best fit algorithm. The actual implementation of `GC_allochblk` is significantly complicated by black-listing issues (see below).

Small blocks are allocated in blocks of size `HBLKSIZE`. Each block is dedicated to only one object size and kind. The allocator maintains separate free lists for each size and kind of object.

In order to avoid allocating blocks for too many distinct object sizes, the collector normally does not directly allocate objects of every possible request size. Instead request are rounded up to one of a smaller number of allocated sizes, for which free lists are maintained. The exact allocated sizes are computed on demand, but subject to the constraint that they increase roughly in geometric progression. Thus objects requested early in the execution are likely to be allocated with exactly the requested size, subject to alignment constraints. See `GC_init_size_map` for details.

The actual size rounding operation during small object allocation is implemented as a table lookup in `GC_size_map`.

Both collector initialization and computation of allocated sizes are handled carefully so that they do not slow down the small object fast allocation path. An attempt to allocate before the collector is initialized, or before the appropriate `GC_size_map` entry is computed, will take the same path as an allocation attempt with an empty free list. This results in a call to the slow path code (`GC_generic_malloc_inner`) which performs the appropriate initialization checks.

In non-incremental mode, we make a decision about whether to garbage collect whenever an allocation would otherwise have failed with the current heap size. If the total amount of allocation since the last collection is less than the heap size divided by `GC_free_space_divisor`, we try to expand the heap. Otherwise, we initiate a garbage collection. This ensures that the amount of garbage collection work per allocated byte remains constant.

The above is in fat an oversimplification of the real heap expansion heuristic, which adjusts slightly for root size and certain kinds of fragmentation. In particular, programs with a large root set size and little live heap memory will expand the heap to amortize the cost of scanning the roots.

(It has been suggested that this should be adjusted so that we favor expansion if the resulting heap still fits into physical memory. In many cases, that would no doubt help. But it is tricky to do this in a way that remains robust if multiple application are contending for a single pool of physical memory.)

Mark phase

The marker maintains an explicit stack of memory regions that are known to be accessible, but that have not yet been searched for contained pointers. Each stack entry contains the starting address of the block to be scanned, as well as a descriptor of the block. If no layout information is available for the block, then the descriptor is simply a length. (For other possibilities, see `gc_mark.h`.)

At the beginning of the mark phase, all root segments are pushed on the stack by `GC_push_roots`. If `ALL_INTERIOR_PTRS` is not defined, then stack roots require special treatment. In this case, the normal marking code ignores interior pointers, but `GC_push_all_stack` explicitly checks for interior pointers and pushes descriptors for target objects.

The marker is structured to allow incremental marking. Each call to `GC_mark_some` performs a small amount of work towards marking the heap. It maintains explicit state in the form of `GC_mark_state`, which identifies a particular sub-phase. Some other pieces of state, most notably the mark stack, identify how much work remains to be done in each sub-phase. The normal progression of mark states for a stop-the-world collection is:

1. `MS_INVALID` indicating that there may be accessible unmarked objects. In this case `GC_objects_are_marked` will simultaneously be false, so the mark state is advanced to
2. `MS_PUSH_UNCOLLECTABLE` indicating that it suffices to push uncollectable objects, roots, and then mark everything reachable from them. `Scan_ptr` is advanced through the heap until all uncollectable objects are pushed, and objects reachable from them are marked. At that point, the next call to `GC_mark_some` calls `GC_push_roots` to push the roots. It the advances the mark state to
3. `MS_ROOTS_PUSHED` asserting that once the mark stack is empty, all reachable objects are marked. Once in this state, we work only on emptying the mark stack. Once this is completed, the state changes to
4. `MS_NONE` indicating that reachable objects are marked.

The core mark routine `GC_mark_from_mark_stack`, is called repeatedly by several of the sub-phases when the mark stack starts to fill up. It is also called repeatedly in `MS_ROOTS_PUSHED` state to empty the mark stack. The routine is designed to only perform a limited amount of marking at each call, so that it can also be used by the incremental collector. It is fairly carefully tuned, since it usually consumes a large majority of the garbage collection time.

The marker correctly handles mark stack overflows. Whenever the mark stack overflows, the mark state is reset to `MS_INVALID`. Since there are already marked objects in the heap, this eventually forces a complete scan of the heap, searching for pointers, during which any unmarked objects referenced by marked objects are again pushed on the mark stack. This process is repeated until the mark phase completes without a stack overflow. Each time the stack overflows, an attempt is made to grow the mark stack. All pieces of the collector that push regions onto the mark stack have to be careful to ensure forward progress, even in case of repeated mark stack overflows. Every mark attempt results in additional marked objects.

Each mark stack entry is processed by examining all candidate pointers in the range described by the entry. If the region has no associated type information, then this typically requires that each 4-byte aligned quantity (8-byte aligned with 64-bit pointers) be considered a candidate pointer.

We determine whether a candidate pointer is actually the address of a heap block. This is done in the following steps:

- The candidate pointer is checked against rough heap bounds. These heap bounds are maintained such that all actual heap objects fall between them. In order to facilitate black-listing (see below) we also include address regions that the heap is likely to expand into. Most non-pointers fail this initial test.
- The candidate pointer is divided into two pieces; the most significant bits identify a `HBLKSIZE`-sized page in the address space, and the least significant bits specify an offset within that page. (A hardware page may actually consist of multiple such pages. HBLKSIZE is usually the page size divided by a small power of two.)
- The page address part of the candidate pointer is looked up in a table. Each table entry contains either 0, indicating that the page is not part of the garbage collected heap, a small integer *n*, indicating that the page is part of large object, starting at least *n* pages back, or a pointer to a descriptor for the page. In the first case, the candidate pointer i not a true pointer and can be safely ignored. In the last two cases, we can obtain a descriptor for the page containing the beginning of the object.
- The starting address of the referenced object is computed. The page descriptor contains the size of the object(s) in that page, the object kind, and the necessary mark bits for those objects. The size information can be used to map the candidate pointer to the object starting address. To accelerate this process, the page header also contains a pointer to a precomputed map of page offsets to displacements from the beginning of an object. The use of this map avoids a potentially slow integer remainder operation in computing the object start address.
- The mark bit for the target object is checked and set. If the object was previously unmarked, the object is pushed on the mark stack. The descriptor is read from the page descriptor. (This is computed from information `GC_obj_kinds` when the page is first allocated.)

At the end of the mark phase, mark bits for left-over free lists are cleared, in case a free list was accidentally marked due to a stray pointer.

Sweep phase

At the end of the mark phase, all blocks in the heap are examined. Unmarked large objects are immediately returned to the large object free list. Each small object page is checked to see if all mark bits are clear. If so, the entire page is returned to the large object free list. Small object pages containing some reachable object are queued for later sweeping.

This initial sweep pass touches only block headers, not the blocks themselves. Thus it does not require significant paging, even if large sections of the heap are not in physical memory.

Nonempty small object pages are swept when an allocation attempt encounters an empty free list for that object size and kind. Pages for the correct size and kind are repeatedly swept until at least one empty block is found. Sweeping such a page involves scanning the mark bit array in the page header, and building a free list linked through the first words in the objects themselves. This does involve touching the appropriate data page, but in most cases it will be touched only just before it is used for allocation. Hence any paging is essentially unavoidable.

Except in the case of pointer-free objects, we maintain the invariant that any object in a small object free list is cleared (except possibly for the link field). Thus it becomes the burden of the small object sweep routine to clear objects. This has the advantage that we can easily recover from accidentally marking a free list, though that could also be handled by other means. The collector currently spends a fair amount of time clearing objects, and this approach should probably be revisited.

In most configurations, we use specialized sweep routines to handle common small object sizes. Since we allocate one mark bit per word, it becomes easier to examine the relevant mark bits if the object size divides the word length evenly. We also suitably unroll the inner sweep loop in each case. (It is conceivable that profile-based procedure cloning in the compiler could make this unnecessary and counterproductive. I know of no existing compiler to which this applies.)

The sweeping of small object pages could be avoided completely at the expense of examining mark bits directly in the allocator. This would probably be more expensive, since each allocation call would have to reload a large amount of state (e.g. next object address to be swept, position in mark bit table) before it could do its work. The current scheme keeps the allocator simple and allows useful optimizations in the sweeper.

Finalization

Both `GC_register_disappearing_link` and `GC_register_finalizer` add the request to a corresponding hash table. The hash table is allocated out of collected memory, but the reference to the finalizable object is hidden from the collector. Currently finalization requests are processed non-incrementally at the end of a mark cycle.

The collector makes an initial pass over the table of finalizable objects, pushing the contents of unmarked objects onto the mark stack. After pushing each object, the marker is invoked to mark all objects reachable from it. The object itself is not explicitly marked. This assures that objects on which a finalizer depends are neither collected nor finalized.

If in the process of marking from an object the object itself becomes marked, we have uncovered a cycle involving the object. This usually results in a warning from the collector. Such objects are not finalized, since it may be unsafe to do so. See the more detailed discussion of finalization semantics.

Any objects remaining unmarked at the end of this process are added to a queue of objects whose finalizers can be run. Depending on collector configuration, finalizers are dequeued and run either implicitly during allocation calls, or explicitly in response to a user request.

The collector provides a mechanism for replacing the procedure that is used to mark through objects. This is used both to provide support for Java-style unordered finalization, and to ignore certain kinds of cycles, *e.g.* those arising from C++ implementations of virtual inheritance.

Generational Collection and Dirty Bits

We basically use the parallel and generational GC algorithm described in "Mostly Parallel Garbage Collection", by Boehm, Demers, and Shenker.

The most significant modification is that the collector always runs in the allocating thread. There is no separate garbage collector thread. If an allocation attempt either requests a large object, or encounters an empty small object free list, and notices that there is a collection in progress, it immediately performs a small amount of marking work as described above.

This change was made both because we wanted to easily accommodate single-threaded environments, and because a separate GC thread requires very careful control over the scheduler to prevent the mutator from out-running the collector, and hence provoking unneeded heap growth.

In incremental mode, the heap is always expanded when we encounter insufficient space for an allocation. Garbage collection is triggered whenever we notice that more than `GC_heap_size`/2 * `GC_free_space_divisor` bytes of allocation have taken place. After `GC_full_freq` minor collections a major collection is started.

All collections initially run interrupted until a predetermined amount of time (50 msecs by default) has expired. If this allows the collection to complete entirely, we can avoid correcting for data structure modifications during the collection. If it does not complete, we return control to the mutator, and perform small amounts of additional GC work during those later allocations that cannot be satisfied from small object free lists. When marking completes, the set of modified pages is retrieved, and we mark once again from marked objects on those pages, this time with the mutator stopped.

We keep track of modified pages using one of three distinct mechanisms:

1. Through explicit mutator cooperation. Currently this requires the use of `GC_malloc_stubborn`.
2. By write-protecting physical pages and catching write faults. This is implemented for many Unix-like systems and for win32. It is not possible in a few environments.
3. By retrieving dirty bit information from /proc. (Currently only Sun's Solaris supports this. Though this is considerably cleaner, performance may actually be better with mprotect and signals.)

Thread support

We support several different threading models. Unfortunately Pthreads, the only reasonably well standardized thread model, supports too narrow an interface for conservative garbage collection. There appears to be no portable way to allow the collector to coexist with various Pthreads implementations. Hence we currently support only a few of the more common Pthreads implementations.

In particular, it is very difficult for the collector to stop all other threads in the system and examine the register contents. This is currently accomplished with very different mechanisms for different Pthreads implementations. The Solaris implementation temporarily disables much of the user-level threads implementation by stopping kernel-level threads ("lwp"s). The Irix implementation sends signals to individual Pthreads and has them wait in the signal handler.

The Irix implementation uses only documented Pthreads calls, but relies on extensions to their semantics, notably the use of mutexes and condition variables from signal handlers. Recent discussions on the net suggest that this could be done more, but not completely, portably.

Both implementations must intercept thread creation and a few other thread-specific calls to allow enumeration of threads and location of thread stacks. This is current accomplished with `# define`'s in `gc.h`.

Comments are appreciated. Please send mail to `boehm@acm.org`