CS 4120/5120 Lecture 38 Memory Management and Garbage Collection 3 May 2018
Lecturer: Andrew Myers

Heap allocation is a necessity for modern programming tasks, and so is automatic reclamation of heap-allocated memory. However, memory management comes with significant costs and has implications for how we generate code.

One way to avoid the cost of heap allocation is to stack-allocate objects. If an object allocated during execution of some function always becomes unreachable when the function returns, the object can be allocated on the function's stack frame. This requires an escape analysis to ensure it is safe. In fact, we can view the escape analysis used to stack-allocate activation records in languages with higher-order functions as an important special case of this optimization.

If an object will never be used again, it is safe to reclaim it. However, whether an object will be used again is an undecidable problem, so we settle for a conservative approximation. An object is *garbage* if it cannot be reached by traversing the object graph, starting from the immediate references available to the current computation(s): registers, stack, and global variables.

A garbage collector automatically identifies and reclaims the memory of these garbage objects.

1 Allocation and memory management

1.1 Linear allocation

To collect garbage, we first have to create it. One strategy for heap management is *linear allocation*, in which the heap is organized so that all used memory is to one side of a `next` pointer. This is depicted in Figure 1. An allocation request of n bytes is satisfied by bumping up the `next` pointer by n and returning its previous value. If the `next` pointer exceeds the `limit` pointer, the non-garbage objects are compacted into the first part of the memory arena so allocation can resume.

Linear allocation makes allocation very cheap, and it also tends to allocate related objects near each in memory, yielding good locality. For functional languages, which do a lot of allocation, it is a good choice. It is commonly used with *copying collectors*, which perform compaction.

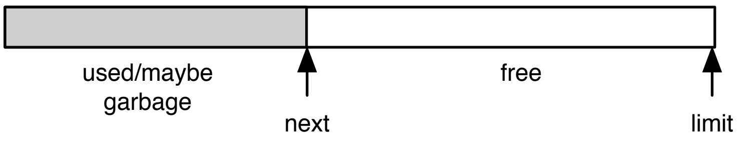

Figure 1: Linear allocation

1.2 Freelist allocation

Another standard organization of memory is to manage free blocks of memory explicitly. The free memory blocks form a linked list in which the first word of each block points to the next free block (it is therefore an *endogenous* linked list). An allocation request is satisfied by following the freelist pointer and finding a block big enough to hold the requested object.

The basic freelist approach wastes space because of two problems: it has *internal fragmentation* in which objects may be allocated a larger block of memory than they requested. And it has *external fragmentation* in which the available memory is split among free memory blocks that are too small to accommodate a request.

We can speed up allocations and reduce internal fragmentation by having multiple freelists indexed by the size of the request. Typically small sizes are covered densely, and sizes grow geometrically after some point. Note, however, that when sizes grow as powers of two, on average about 30% of the space is wasted.

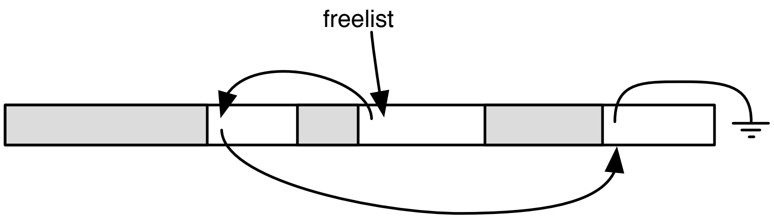

Figure 2: Freelist. Allocated objects shown in gray.

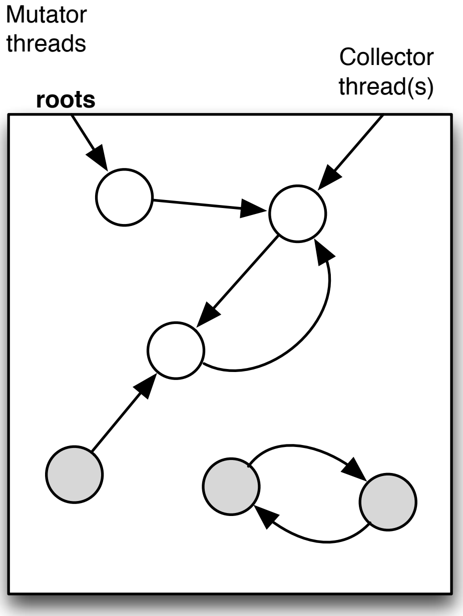

Figure 3: Mutator and GC. Garbage objects shown in gray.

External fragmentation can be reduced by merging adjacent free blocks. This requires additional space for each memory block to keep track of its neighbors. The space may be in the memory block itself, or in an external data structure that keeps track of whether neighbors are in use (e.g., as a bitmap). The Berkeley allocator's "buddy system" is a neat example of the latter, with power-of-two sized memory blocks that can be merged to form a block of the next size up. The standard Linux `malloc` does not use a buddy system, however.

2 Automatic garbage collection

These days we expect the language to collect garbage for us. This is more complex than we might expect, since modern language implementations contain sophisticated garbage collectors. Garbage collectors can do a better job if they can take advantage of help from the compiler, so it is worth understanding how GC and compilers interact.

An ideal garbage collector would have the following attributes:

- Fully automatic.
- Low overhead in time and space.

- *Pause times* in which the program waits for the collector should be short.
- Safe: only garbage is reclaimed.
- Precise: almost all garbage is collected, and reasonably soon.
- Parallel: able to take advantage of additional cores.
- Simple.

Abstractly, we can think of a garbage collector as a thread or threads running perhaps concurrently with the actual compute thread(s). This is shown in Figure 3. We refer to the threads doing real computations as the *mutator*, since they are changing the object graph that the garbage collector is trying to manage.

3 Reference counting

A standard (and often incorrectly implemented) approach to automatic memory management is *reference counting*. This is a simple local technique for collecting garbage. It is usually avoided because of its high runtime overhead, though recent work has argued that its overhead can be made comparable with other GC techniques, and it seems to be coming back into vogue.

Each object has a reference count approximating the number of incoming pointers to the object. For example, in Figure 3, the leftmost gray object has no incoming pointers and a reference count of zero. The other two objects have a reference count of one each. This points out the big problem with reference counting: it does not deal with cycles in the object graph, and must be combined with another garbage collection method if cycles are to be reclaimed.

When a new reference is created to an object, its reference count is incremented; when a reference is destroyed, its reference count is correspondingly decremented. This means that a simple assignment like `x=y` requires incrementing the reference count of `y` and decrementing the reference count of the object `x` used to point to.[1] This is a lot of overhead on top of what was formerly a simple `mov` between two registers.

When the reference count of an object goes to zero, it is ready to be reclaimed. It is tempting to release its memory immediately and (recursively) decrement the reference counts for all outgoing pointers, but this leads to long pause times. The correct implementation is to push such objects onto a queue, to be processed later. On an allocation request for n bytes, objects totaling n bytes in size should be popped from the queue (if possible) and reclaimed after the reference counts for all successor objects have been decremented appropriately. This deferred decrementing means that the amount of work done by the reference counting machinery at any given moment is relatively small. Popping at least as many bytes from the queue as are requested ensures that the queue doesn't build up, trapping memory.

Reference counting generates a lot of write traffic to memory and hurts performance. This traffic is particularly a problem in the case of objects that are shared between multiple cores. It is critical that reference counts be updated atomically and that all cores see increments to reference counts.

The overhead of writes to update reference counts can be reduced by avoiding updating reference counts. The key to this optimization is to notice that it is not necessary that the reference count exactly match the number of incoming references at all times. It is enough to enforce a weaker invariant:

- For safety, the reference count should only be decremented to zero if there are no incoming references.
- For precision, an object with no incoming references should eventually have a reference count of zero.

This weaker invariant implies that it is not necessary to update the reference count to an object over some scope during which an additional reference is known to exist, if it is known that another reference to the object exists and will prevent its count from going to zero. Also, it is possible to defer decrementing the reference count of an object, perhaps until the time of a later increment, in which case both updates can be eliminated. And multiple increments and decrements to the same object can be consolidated. These are all helpful optimizations when generating code that uses reference counting. Researchers have reported reducing reference counting overhead by roughly 80% through such optimizations.

[1] Note that doing the decrement first could be a disaster if `x` and `y` happen to be aliases!

Another interesting optimization based on reference counting is to reuse existing allocated space for new allocations. If it can be determined statically that the number of references to a particular object is zero at a certain point in the code, that allocated object can be reused for other allocations that happen in the same scope.

4 Garbage collection via traversal

To identify cycles, we need an algorithm that traverses the object graph. Such algorithms, of which there are many, can be viewed as instances of Dijkstra's tricolor graph traversal algorithm.

In this algorithm, there are three kinds (or colors) of objects. *White* objects are the unvisited objects that have not yet been reached in the traversal. *Gray* objects have been reached, but may have successor objects that have not been reached. *Black* objects are completed objects. The key invariant of the algorithm is that black objects can never point to white objects. The algorithm starts with all nodes colored white. It then proceeds as follows:

1. The roots of garbage collection are colored gray.
2. While a gray object g exists:
   - Color any white successors of g gray.
   - Color g black.
3. All objects are now black or white, and by the invariant, white objects are not reachable from black objects, and are therefore garbage. Reclaim them.
4. Optionally compact black objects.

5 Finding pointers

To be able to traverse objects in the heap, the collector needs to be able to find pointers within objects, on the stack, and in registers, and to be able to follow them to the corresponding heap objects. Several techniques are used.

5.1 Tag bits

In this approach, some number of bits in each word are reserved to indicate what the type of the information is. At a minimum, one bit indicates whether the word is a pointer or not. It's convenient to use the low bit and set it to one for pointers, zero for other information such as integers. With this representation, a number n is represented as $2n$, allowing addition and subtraction to be performed as before. Accesses via pointers must have their offsets adjusted. For example, the address 0x34 (hexadecimal) would be represented as 0x35. A memory operand `4(%rsi)` would become `3(%rsi)` to compensate for the one's bit being set in `%rsi`.

5.2 Computed GC information

To avoid using space in each word, an alternative is to store elsewhere the information about which words are pointers. In an OO language, this information can be included in the dispatch table and shared among all objects of the same type. The compiler also needs to compute which words on the stack are pointers and record it for use by the garbage collector. The program counter for each stack frame is saved in the next stack frame down, so it is possible to use the program counter to look up the apropriate stack frame description for that program point. Note that in general the stack frame description may change during a procedure, so there may be multiple such table entries per procedure.

5.3 Conservative GC

Conservative garbage collection, of which the Boehm–Demers–Weiser (BDW) collector is probably the best-known example, treats everything that might be a pointer as if it were one. Thus, a conservative collector can even be used with a programming language like C that has no run-time support for garbage collection. The reason this works well is that integers used in programs are usually small and can't be confused with pointers if the program runs high enough up in virtual memory; in any case, the effect of treating a non-pointer as a pointer is that garbage might not be reclaimed. This is usually not a problem.

Conservative GC has the advantage that it requires no compiler or runtime help (though these collectors can do a better job with compiler help).

Because pointers or apparent pointers, can go into the middle of objects, the heap needs more data structure overhead to supporting finding the beginning and end of an object given a pointer somewhere into it, and more careful logic to avoid dereferencing things that are not really pointers.

6 Mark and sweep

Mark and sweep is a classic GC technique. It uses a depth-first traversal of the object graph (the mark phase), followed by a linear scan of the heap to reclaim unmarked objects (the sweep). The mark phase is an instance of the tricolor algorithm, implemented recursively:

```
visit(o) {
    if (o is not white) return
    color o gray
    foreach (o’>o) visit(o’)
    color o black
}
```

The marked objects are the non-white objects; gray objects are those whose subtrees of the traversal is not yet complete. Once the mark phase is done, the sweep phase scans through the heap to find unmarked (white) blocks. These blocks are added to the freelist.

One problem with mark-and-sweep is that when the recursive traversal is implemented naively, the recursion depth and hence the stack space is unbounded. The solution is to convert the algorithm into an iterative one. On returning from a call to `visit`, the state to be restored consists of the object being scanned in the `foreach` loop. The pointer to this predecessor object can be stored into the current object, which is called `pred` in the following iterative algorithm:

```
while (o.pred != null) {
    color o gray
    foreach (o’>o) {
        if (o’ is white) {
            o’.pred = o
            o = o’
            continue
        }
    }
    color o black
    o = o.pred
}
```

To avoid the overhead of storing this extra pointer, the Deutsch–Waite–Schorr pointer reversal technique overwrites the pointer to the next object `o’` with this predecessor pointer, and restores it when traversal of `o’` completes. The only extra storage needed per object is enough bits to indicate which pointer within the object is being traversed (and hence has been overwritten). The visit() procedure is modified to return the

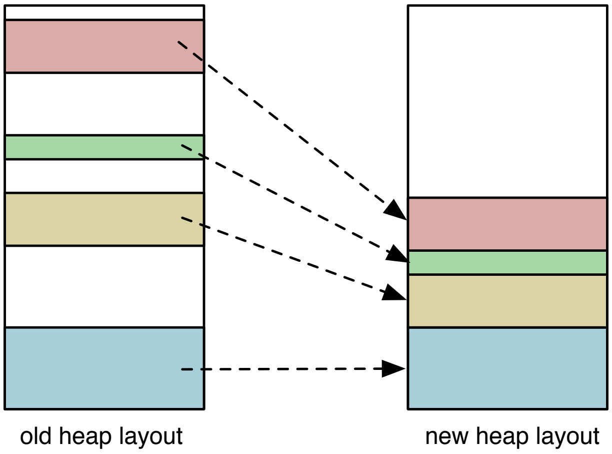

Figure 4: Compacting the heap

current object, so that the overwritten pointer to it can be restored in the caller. However, pointer reversal makes the heap unusable for computation. The mutator must be paused during the mark phase.

Mark-and-sweep is famous for introducing long garbage collection pauses, though incremental versions exist.

7 Mark and compact

Compacting the heap improves locality and eliminates external fragmentation. During the sweep phase, it is possible to compute and record new compacted locations for each of the live objects. A second pass over the heap can then adjust all pointers stored in objects to point to the new locations of their referents. The objects can then be copied down to their new locations in a third pass. Of course, three passes over the heap is expensive and poor cache performance can be expected. Copying can be done at the same time as updating all the pointers if the new object locations are stored elsewhere than in the objects themselves, eliminating one pass.

8 Copying collection

Copying collection can be a better way to achieve efficient compacting collection. The idea is to split memory into two equal-sized portions; the mutator uses only half the memory at any given time. Linear allocation is used in the half of memory that is active. When the `next` pointer reaches the `limit` pointer, the half that is in use becomes the *from-space* and the other half becomes the *to-space*. The from-space is traversed, copying reached objects into to-space. Objects that are copied to the to space must have each pointer rewritten to point to the corresponding to-space object; this is achieved by storing a forwarding pointer in the object, pointing from from-space to the corresponding to-space object.

Depth-first traversal can be used to perform copying, but has the problem of unbounded stack usage. An alternative that does not require stack space is Cheney's algorithm, which uses breadth-first traversal. As shown in Figure 5, two pointers are maintained into the to-space. The first is the `scan` pointer, which separates objects that have been copied and completely fixed to point to to-space (black objects) from objects that have been copied but not yet fixed (gray objects). The second pointer is the `next` pointer, where new objects are copied. The collector starts by copying root objects to to-space. It then repeatedly takes the first object after the `scan` pointer and fixes all of its pointers to point to to-space. If its pointers point to an object that has already been copied, the forwarding pointer is used. Otherwise, the object is copied to to-space at `next`. Once all the pointers are fixed, the object is black, and `scan` pointer is adjusted to point to the next

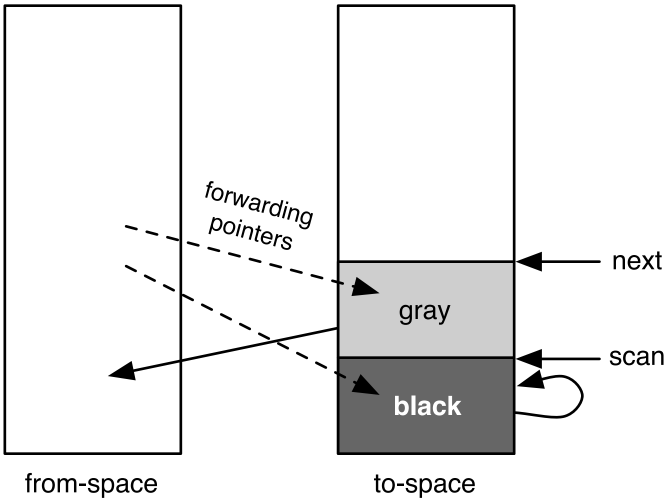

Figure 5: Copying collection with breadth-first traversal

object. When the `scan` and `next` pointers meet, any uncopied objects left in from-space are garbage. On the next garbage collection, the roles of from-space and to-space swap. One nice property of this algorithm is that the work done is proportional to the number of live objects.

Breadth-first traversal is simple but has the unfortunate property of destroying locality in the to-space. A hybrid approach is to augment breadth-first traversal to use depth-first traversal. As each object is copied to to-space, a depth-first traversal *to limited depth* is performed from the object, copying all reached objects. This helps keep objects in memory near objects they point to.

9 Concurrent and incremental collection

To avoid having long pause times, it is helpful to have the collector doing work at the same time as the mutator. A concurrent collector runs concurrently with the mutator and therefore needs to synchronize with the mutator to avoid either breaking the other. An incremental collector is one that can run periodically when permitted by the mutator, and make some progress toward garbage collector but without completing a GC. Typically an incremental collector would run on each allocation request.

The danger of both concurrent and incremental collection is that invariants of the collector may be broken by the mutator. In particular, the key invariant of the tricolor algorithm—that black objects cannot point to white objects—might be broken if the mutator stores a white object the collector hasn't seen into a black object that the collector thinks it is done with.

There are two standard ways to prevent this invariant from being broken by the mutator: *read barriers* and *write barriers*, which are generated by the compiler as part of code generation.

- **Read barriers.** The mutator checks for white objects as it reads objects. If a white object is encountered, it is clearly reachable, and the mutator colors the object gray so that it is safe to store it anywhere. Since the collector must know about all gray objects, this means in practice that the reached object is put onto a queue that the collector is reading from. The objects considered gray include this queue.

  An example of this approach is Baker's algorithm for concurrent copying collection. Rather than pausing the mutator during all of collection, the mutator is paused only to copy the root objects to to-space. The mutator then continues executing during collection, using only to-space objects. However, the mutator may follow a pointer from a gray object back to from-space. The read barrier then checks for from-space pointers. A from-space pointer to an already-copied object is corrected to a to-space pointer using the forwarding pointer; a from-space pointer to an object not copied yet triggers copying.

- **Write barriers.**

  With write barriers, the mutator checks directly whether it is storing a pointer into black objects; if it is, the object is colored gray, causing the collector to rescan the object. This approach requires less tight coordination between the mutator and the collector, and the barrier is less expensive because writes are less frequent than reads, and because the write barrier only affects pointer updates, not other types such as integers.

  This approach was pioneered by the "replication-based copying" garbage collection technique of Nettles and O'Toole. The mutator runs in *from-space*, unlike in Baker's algorithm, and allocates new objects there. Updated objects in from-space might contain pointers to objects that need to be saved, so the updates need to be propagated to to-space and those pointers need to be followed before swapping the spaces. Therefore, these objects are added to a queue that the garbage collector processes before declaring GC complete. Once GC completes, the mutator and collector synchronized so the roots can be switched to point to to-space. Since the mutator runs on from-space objects, an extra header word is needed per object to store the forwarding pointer to to-space.

Using either read or write barriers, the pause time caused by incremental garbage collectors can be reduced to the point where it is imperceptible.

10 Generational garbage collection

Most objects die young. This observation motivates generational garbage collection. Effort expended traversing long-lived objects is likely to be a waste. It makes sense to instead focus garbage collection effort on recently created objects that are likely to be garbage.

Generational garbage collection (aka generational scavenging) assigns assign objects to distinct *generations*, with the collector usually working on just the youngest generation. Objects that survive enough collections are promoted to an older generation. The analogy to the academic tenure process has been noted.

Generational collection of the youngest generation works particularly well with copying collection. Since the youngest generation is small, the 2× space overhead is not a problem. Since copying collection does work proportional to the live objects, and the youngest generation is likely mostly dead, collection is also fast. The other benefits of copying collection: good locality from compaction, and fast linear allocation. For older generations, copying collection is usually a bad idea: it causes needless copying, increases the total memory footprint of the system, and does not bring significant compaction benefits.

When doing a generational collection, pointers from older generations into the generation(s) being collected are treated as roots. Fortunately, it is atypical for older generations to point to younger generations, because the older-generation object is pointing to an object that was created later. These older-generation objects are called the *remembered set* for the younger generation. Objects become part of the remembered set only when they are updated imperatively to point to the younger generation. The remembered set is therefore tracked by using a write barrier to detect pointer updates that affect the remembered set.

To reduce the overhead of tracking the remembered set, *card tables* are often used. The insight is that it is not necessary to track the remembered set precisely—any superset of the remembered set suffices for safety, though we would like to avoid including very many extra objects. A card table is a bit vector mapping older-generation heap regions to bits. A 0-bit in the vector means no remembered-set objects are in that heap region; a 1-bit means there are some remembered-set objects in the region. The card table is a compact representation of the remembered set and it can updated very quickly when necessary. When collecting the younger generation, all objects in the heap sections marked "1" must be scanned, conservatively, because they might contain a pointer to the younger generation. Card table bits can be cleared during this scan if no younger-generation pointers are found.

Another approach to tracking the remembered set (and implementing other write barriers) is to use the virtual memory system to trap writes to older generations. The pages containing older generations are marked read-only, triggering a page fault when a write to an old-generation object is attempted. The page fault handler adds the old-generation object to the remembered set if it is updated to point to a younger-generation object.

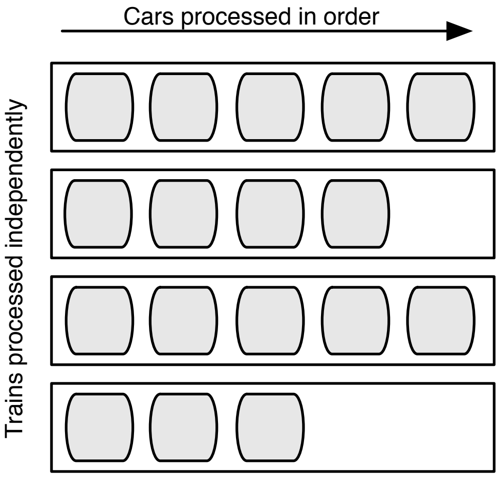

Figure 6: The train algorithm

11 Mature object GC

In long-lived applications, the oldest generation often becomes large, so garbage-collecting it is expensive. But garbage-collecting any subregion of it will fail to collect cycles that cross between regions. The *train algorithm* of Hudson and Moss is an incremental way to collect mature objects with good performance and precision, while eventually collecting cycles. It is also relatively simple to implement.

The heap is divided into multiple *trains* containing some number of *cars*. Cars are parts of the heap that are garbage-collected together. They are the units of GC. However, each train can be garbage-collected independently. Given a train to be garbage-collected, the first test is whether it has any incoming edges. If not, the whole train is garbage-collected. Of course, keeping track of incoming edges requires a remembered set and therefore a write barrier.

Given that the train has some incoming edges, the algorithm selects the first car from the train. Using the remembered set for the car, all objects in the car that are referenced externally are evacuated to the last car of a train containing a reference to the object (which may or may not be the same train). If the destination train's last car is too full, a new car is created there as needed.

In addition, a traversal is done from these evacuated objects to find all other objects in the same car reachable from them; these other objects are evacuated to the same train (and car, if possible). After evacuating objects, all remaining objects are garbage. The car is destroyed along with its remaining objects.

Since only one car is collected at a time, GC pause time is very short.

By moving objects to the same car as their predecessors, the train algorithm tends to migrate cycles into the same train car. This allows them to be garbage-collected efficiently.