Chapter 2

Garbage Collection

Garbage collection is an essential component of modern high-level languages, enabling strong type-safety and memory-safety guarantees. However, garbage collection has the potential to adversely affect performance, in terms of *throughput*, *responsiveness*, and *predictability*. This chapter provides an overview of garbage collection, covering fundamental algorithms and mechanisms. The focus is on garbage collection approaches that have the potential to address *all* of these performance criteria simultaneously, yielding predictable, highly-responsive, high-throughput systems.

Section 2.1 provides a brief overview of garbage collection, defines necessary terminology, and introduces fundamental *algorithms* and *mechanisms*. Section 2.2 discusses the performance requirements of low-level programs with respect to garbage collection. Sections 2.3 and 2.4 then describe garbage collection techniques that have the potential to meet the particular performance requirements of low-level programming. Section 2.3 discusses work on *incremental* and *concurrent* tracing garbage collection: techniques that allow garbage collection to proceed alongside application activity. Section 2.4 then describes an alternative approach based on reference counting, an inherently incremental approach to garbage collection often used in low-level programming.

2.1 The Anatomy of a Garbage Collector

This section provides a brief overview of garbage collection terminology, algorithms, and mechanisms. For a more complete discussion of the fundamentals of garbage collection see "Garbage Collection: Algorithms for Automatic Dynamic Memory Management" [Jones and Lins, 1996], and "Uniprocessor Garbage Collection Techniques" [Wilson, 1992].

Programs require data to execute, and this data is typically stored in memory. Memory can be allocated statically (where memory requirements are fixed ahead-of-time), on the stack (tightly binding the lifetime of the data to the currently executing method), or *dynamically*, where memory requirements are determined during execution—potentially changing between individual executions of the same program. This dynamically allocated *heap* memory can be explicitly managed by the program (through primitives such as the C functions `malloc` and `free`), or it can be

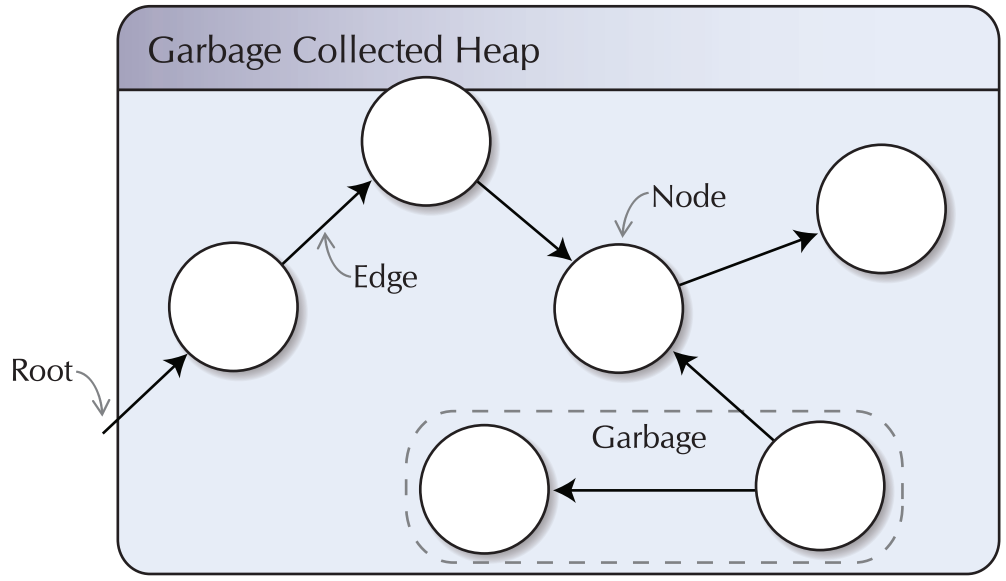


Figure 2.1: An object graph, showing *nodes*, *edges*, and a *root*.

automatically managed through the use of a garbage collector.

Garbage collection takes the burden of explicitly managing memory away from the programmer. While there are many cases in which this burden is insignificant, complex systems with large, shared data-structures make the explicit management of memory both an onerous and error-prone task. The need to manage memory explicitly also compromises software design, forcing additional communication between modules in order to ensure that a *global* consensus is reached before any shared data is freed.

The role of the garbage collector is to reclaim memory that is no longer required by the application. To assist the discussion of garbage collection, we will view all objects in memory as a *directed graph* as shown in Figure 2.1. Objects are represented as *nodes*, and references between objects are represented as directed *edges*. There are also edges originating from outside the object graph—such as values held in program variables—which are known as *roots*. In accordance with the terminology of Dijkstra et al. [1978], application threads that manipulate this object graph (by allocating new objects and changing the set of edges) are known as *mutators*, while threads that perform garbage collection work are known as *collectors*.

Determining precisely when an object will no longer be accessed is difficult in general, so garbage collectors rely on a conservative approximation based on *reachability*. Any object that is *unreachable*—that is, there is no path from a root to the node over the edges in the graph—can never be accessed again by the application, and may therefore be safely reclaimed.

2.1.1 Taxonomy of Garbage Collection Algorithms

Memory management approaches can be categorized based on how they solve three key sub-problems: object *allocation*, garbage *identification*, and garbage *reclamation*. Naturally, approaches to each of these sub-problems have a synergistic relationship with solutions to other sub-problems. Many memory management approaches are hybrids, drawing on several approaches to each of these sub-problems.

2.1.1.1 Object Allocation

There are two fundamental techniques used for object allocation—*bump pointer* allocation, and *free list* allocation.

Bump pointer allocation. Under bump pointer allocation (see Figure 2.2), memory is allocated by running a cursor across memory, *bumping* the cursor by the size of each allocation request. Bump pointer allocation is simple and fast at allocation time, but provides no facility for incremental freeing. Given the simplicity of the approach, the design space for bump pointer allocation is quite restricted. One key design consideration is the approach used to allow parallel allocation. Bump pointer allocation schemes generally perform synchronized allocation of larger chunks [Garthwaite and White, 1998; Alpern et al., 1999; Berger et al., 2000], which are then assigned to a single thread—allowing fast, unsynchronized bump pointer allocation within the chunk.

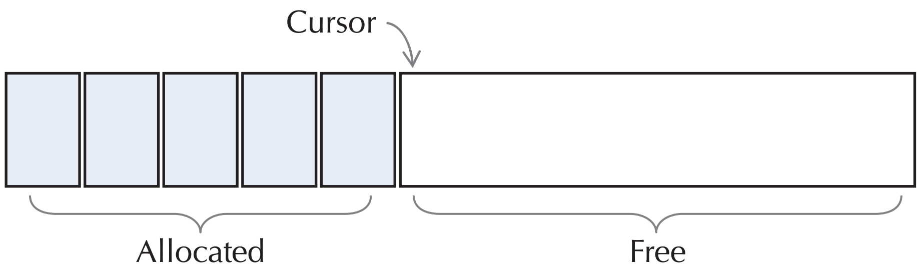


Figure 2.2: Bump pointer allocation.

Free list allocation. Under free list allocation (see Figure 2.3), memory is divided into cells, which are then maintained in a list—the free list. Throughout this thesis, the free list structure of most interest is the *segregated* free list, such as that described by Boehm and Weiser [1988].[1] The segregated free list scheme (described as a *two-level* allocation scheme by Jones and Lins [1996]) attempts to balance the concerns of fragmentation and throughput performance. In a segregated free list scheme an allocator manages *multiple* free lists, each containing a list of empty cells of a single fixed

[1]The design space for free list allocation is large; refer to Jones and Lins [1996] or Wilson et al. [1995] for a more complete discussion.

size. Because each list contains cells of a fixed size, allocation is fast and no searching is required. Memory is divided up into larger *blocks*, each containing cells of a fixed size. Blocks are then managed on a block free list, with empty blocks available for use by *any* size class. This structure addresses fragmentation for most programs, failing only when a program: 1) allocates many objects of a given size class; 2) keeps a small fraction alive (pinning down many blocks); and then 3) changes allocation patterns to allocate many objects of *different* size classes. This pathology is generally rare and can be addressed through some form of copying collection.

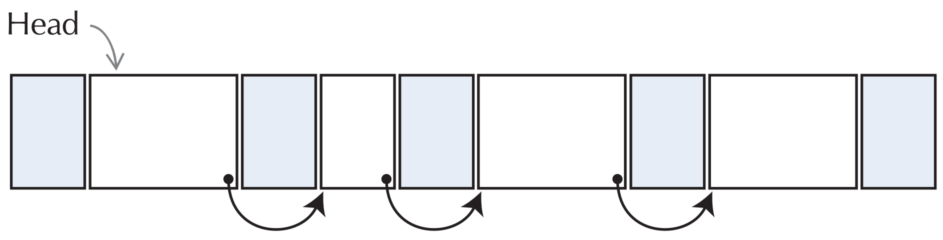


Figure 2.3: Free list allocation.

2.1.1.2 Garbage Identification

There are two fundamental techniques for identifying garbage data: *reference counting* and *tracing*. Each of these techniques forms the basis for one or more of the canonical garbage collection algorithms described in more detail in Section 2.1.2.

Reference counting. This method *directly* identifies garbage. Each object has a *reference count*, which keeps track of the number of references (incoming edges) to that object in the object graph. When a reference count falls to zero, the associated object can be considered garbage.

Tracing. This method *indirectly* identifies garbage by directly identifying all live objects. Tracing involves performing a *transitive closure* across some part of the object graph—visiting all objects transitively reachable from some set of root edges—identifying each visited object as live. All objects that were *not* visited during the trace are identified as garbage.

2.1.1.3 Garbage Reclamation

Once objects have been allocated, and those that need to be collected have been identified, there are several techniques that can be used to reclaim the space.

Direct to free list. For direct garbage collection approaches (e.g., reference counting) it is possible to directly return the space for objects to a free list.

Evacuation. Live objects can be *evacuated* from a region of memory, which once emptied of live data, may be reclaimed in its entirety. This approach requires another region of memory into which the live objects can be copied. Evacuation can be particularly effective when there are very few survivors, and naturally aligns itself with tracing as the approach for identification.

Compaction. Compaction rearranges the memory within the region *in-place* to allow future allocation into the region. A classic example is sliding compaction [Styger, 1967; Abrahams et al., 1966] where all live data is compressed into a contiguous chunk of used memory, leaving a contiguous chunk of memory free for future allocation.

Sweep. A sweep is a traversal over allocated objects in the heap, freeing the space associated with objects that have been identified as garbage. Some form of sweep is required by many tracing approaches, because garbage is not identified directly. While sweep generally operates over individual free list cells, it is also possible to use the sweep approach on larger regions of memory.

2.1.2 Canonical Algorithms

This section briefly introduces canonical algorithms that cover the design space laid out above.

2.1.2.1 Reference Counting

One of the classic forms of garbage collection is reference counting [Collins, 1960]. Recall from above that reference counting works by keeping track of the number of incoming edges—or references—to each node in the object graph. When this count drops to zero, the object is known to be unreachable and may be collected. Figure 2.4 shows an object graph with reference counts calculated, and also demonstrates the fundamental weakness of reference counting: *cyclic garbage*. The objects X and Y are clearly unreachable (there is no path to either of them from any root) but they will never be collected because they still hold counted references to each other. Unless additional work is performed to identify and collect it, cyclic garbage can cause memory leaks.

Reference counting is of particular interest for high-level low-level programming, and is discussed in detail in Section 2.4. It is inherently incremental, and uses object-local information only, rather than requiring any global computation. Explicit reference counting—where the programmer manually manipulates reference count information—is a proven approach for low-level programming, extensively used for managing data structures in low-level software written in languages such as C and C++.

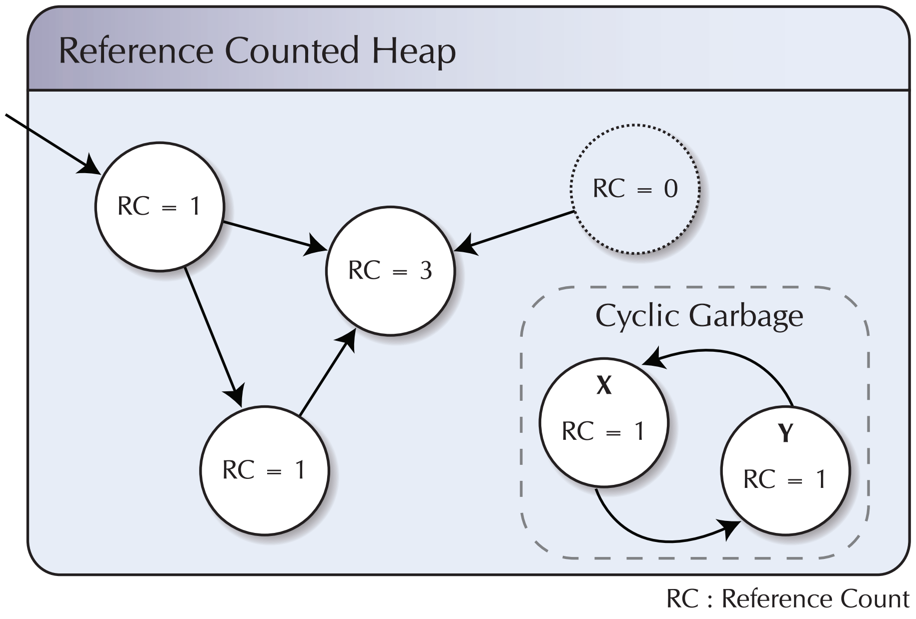


Figure 2.4: An object graph showing *reference counts* and *cyclic garbage*.

2.1.2.2 Mark-Sweep

Mark-sweep collection [McCarthy, 1960] is a *tracing* collection approach that runs in two simple phases:

1. A *mark* phase, which performs a transitive closure over the object graph, *marking* objects as they are visited.

2. A *sweep* phase, where *all* objects in the heap are checked, and any that are not marked are unreachable and may be collected. It is possible for part of this sweeping phase to be performed during execution, a technique called *lazy sweeping* which reduces garbage collector time, and can actually improve *overall* performance, due to the sweep operation and subsequent allocations being performed on the same page, improving cache behavior.

Mark-sweep collection is efficient at collection time, but forces the mutator to allocate objects in the discovered holes surrounding live objects. This can result in lower allocation performance, as well as generally exhibiting poor locality of reference due to objects being spread over the heap. Over time, mark-sweep can also encounter problems with fragmentation—even though the sum total of available memory may be sufficient, an empty, *contiguous* region of sufficient size may not be available.

2.1.2.3 Semi-Space

Semi-space collection is also based on tracing, but uses a very different approach to reclaim memory. Memory is logically divided into two equally sized regions.

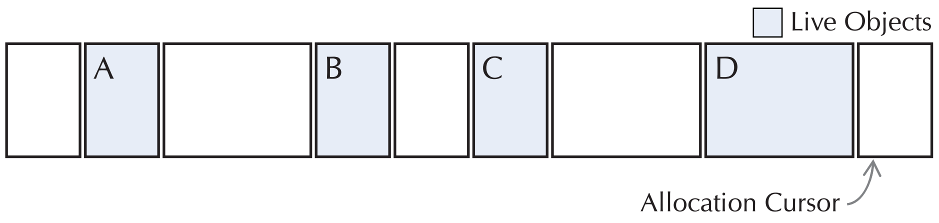
During program execution one region contains objects, while the other region is empty. When garbage collection is triggered, the region containing objects is labeled as the *from-space* and the empty region is labeled as the *to-space*. Garbage collection proceeds by performing a transitive closure over the object graph, copying all nodes encountered in from-space into to-space—updating all edges to point to the copied objects in to-space. At the end of collection all reachable objects have been copied out and saved, and all that remains in the from-space is unused. This from-space is now considered empty, and the to-space contains all live objects: a reversal of the roles of the two regions prior to the collection. Initial implementations of copying collectors used a recursive algorithm [Minsky, 1963; Fenichel and Yochelson, 1969] but a simple iterative algorithm was later introduced by Cheney [1970]. In comparison to mark-sweep collection, semi-space collection:

- makes less efficient use of memory as it must hold 50% percent of total memory as a *copy reserve* to ensure there is space to copy all objects in the worst case;
- can be more expensive at collection time because all live objects must be copied;
- can be *cheaper* at collection time if very few objects survive, because no sweep phase is required;
- can utilize efficient bump pointer allocation, because free memory is always maintained as a contiguous block; and
- has less problems with fragmentation, because live objects are copied into a contiguous chunk of memory.

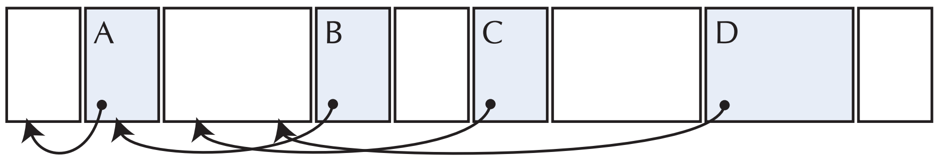
2.1.2.4 Mark-Compact

Mark-compact collection aims to combine the benefits of both semi-space and mark-sweep collection. It addresses fragmentation often seen in mark-sweep by *compacting* objects into contiguous regions of memory, but it does so *in place* rather than relying on the large copy reserve required by semi-space collection. While this in-place transition saves space, it typically involves significant additional collection effort. This is because additional care must be taken to ensure that the target location of a copied object does not contain live data. A simple form is *sliding* compaction, which logically compresses all live objects—in allocation order—into a single contiguous chunk. A simple sliding compaction algorithm—known as the LISP-2 algorithm [Styger, 1967]—proceeds as follows, with the state at the conclusion of key phases shown in Figure 2.5:

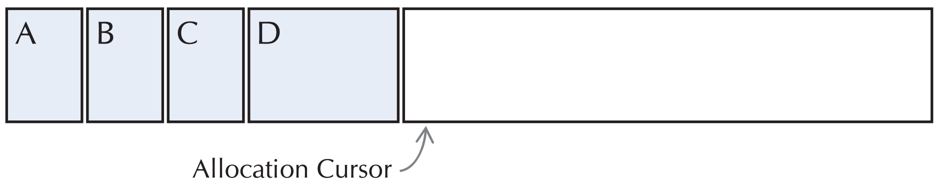
1. A *mark* phase, which performs a transitive closure over the object graph, *marking* objects as they are visited.
2. A *compute-forwarding-pointers* phase, where objects are processed *in address order* and the future location of each marked object is calculated. The calculation occurs by simply incrementing a cursor by the size of each live object as it is encountered.



(a) After the *mark* phase.



(b) After the *compute-forwarding-pointers* phase.



(c) After the *relocate* phase.

Figure 2.5: Sliding compacting collection.

3. A *forwarding* phase, where all pointers are updated to reflect the addresses calculated in the previous phase. Note that after this phase all references point to *future* locations of objects, rather than the *current* location.

4. A *relocation* phase, where objects are copied to their target locations in address order. The address order is important as it ensures that the target location does not contain live data.

The additional phases of simple compaction algorithms make them significantly more expensive than simple mark-sweep or semi-space collection. While there have been many attempts to reduce this cost—by optimizing the phases, reducing the number of phases, or by implementing the phases as operations on compact representations of the heap [Kermany and Petrank, 2006]—compaction is rarely used as the sole collection strategy in a high-performance system. Compaction is, however, commonly combined with mark-sweep collection (to provide a means to escape fragmentation issues), and is often used alongside semi-space collection [Sansom, 1991] to allow execution to continue when memory is tight.

2.1.2.5 Mark-Region

Mark-Region is a collection approach that combines contiguous allocation and non-copying tracing collection. The motivation of this approach is to combine the mutator performance of semi-space with the collection performance of mark-sweep. In terms of allocation, mark-region is similar to semi-space, with objects allocated into contiguous *regions* of memory using a bump pointer. In terms of collection, mark-region is similar to mark-sweep, but sweeps entire *regions*; regions with no reachable objects are made available again for contiguous allocation. Immix [Blackburn and McKinley, 2008] provides the first detailed analysis and description of a mark-region collector, although a mark-region approach was previously used in JRockit [Oracle] and IBM [Borman, 2002] production virtual machines. A mark-region collection proceeds as follows:

1. A *mark* phase, which performs a transitive closure over the object graph, marking *regions* which contain live objects as each object is visited.

2. A *sweep* phase, where *regions* that were not marked in the previous phase are made available for future contiguous allocation.

The mark-region approach is susceptible to issues with fragmentation, because it may not be possible to discover large contiguous blocks to allow efficient bump pointer allocation. To combat this, mark-region collectors often employ techniques to relocate objects in memory to reduce fragmentation. The JRockit collector performs compaction of a fraction of the heap at each collection, the IBM collector performs whole heap compaction when necessary, and Immix performs lightweight defragmentation as required when memory is in demand.

2.1.3 Generational Collection

Generational garbage collection [Lieberman and Hewitt, 1983; Moon, 1984; Ungar, 1984] is perhaps the single most important advance in garbage collection since the first collectors were developed in the early 1960s. The generational hypothesis states that most objects have very short lifetimes. Generational collectors are optimized for when this hypothesis holds, and thereby attain greater collection efficiency by focusing collection effort on the most recently allocated objects.

Generational collectors partition the heap into *generations* based on allocation age. This thesis considers only the basic form of generational collection, where the heap is divided into two generations: the *nursery*—containing the most recently allocated set of objects—and the *mature* area—containing all other objects. In order to independently collect the nursery, generational collectors must remember all pointers from the mature space into the nursery. This can be achieved by building a *remembered set* of pointers created into the nursery, or by remembering regions of the mature space (usually referred to as *cards*) that contain nursery pointers, and must be scanned at nursery collection time. References from the mature space into the nursery, in combination with any other roots (e.g., program variables), then provide the starting point for a transitive closure across all live objects within the nursery. A partial copying collection can be performed during this closure, with live objects evacuated from the nursery into the mature space. When the generational hypothesis holds, this collection is very efficient because only a small fraction of nursery objects must be copied into the mature area.

2.1.4 Barriers

Barriers are operations that are injected into mutator code surrounding mutator operations that may affect the garbage collector. Barriers are most commonly inserted on read and write operations, and are essential tools for more powerful garbage collection algorithms. In reference counting collectors, reference write barriers can be used to perform necessary reference count increments and decrements. Generational collectors may also rely on reference write barriers to intercept pointers created from mature objects to nursery objects. Concurrent and incremental garbage collectors may make extensive use of read and write barriers to keep them informed of potentially destructive changes, and to ensure that mutators are operating on and updating the appropriate data.

Given the algorithmic power of barriers, it is essential that high-performance barrier operations be available in order to judge the true cost of a given garbage collection approach. The performance of barriers is a complex interaction of multiple factors including the operation itself, how it is optimized (e.g., what portions are inlined), the behavior of individual applications, and the underlying architecture [Hosking et al., 1992; Blackburn and McKinley, 2002; Blackburn and Hosking, 2004]. It is also possible to elide barriers to improve performance by identifying cases in which the barrier operation is redundant [Vechev and Bacon, 2004].