Run-Time Environments/Garbage Collection

Sandro Spina

Department of Computer Science, Faculty of ICT

January 5, 2014

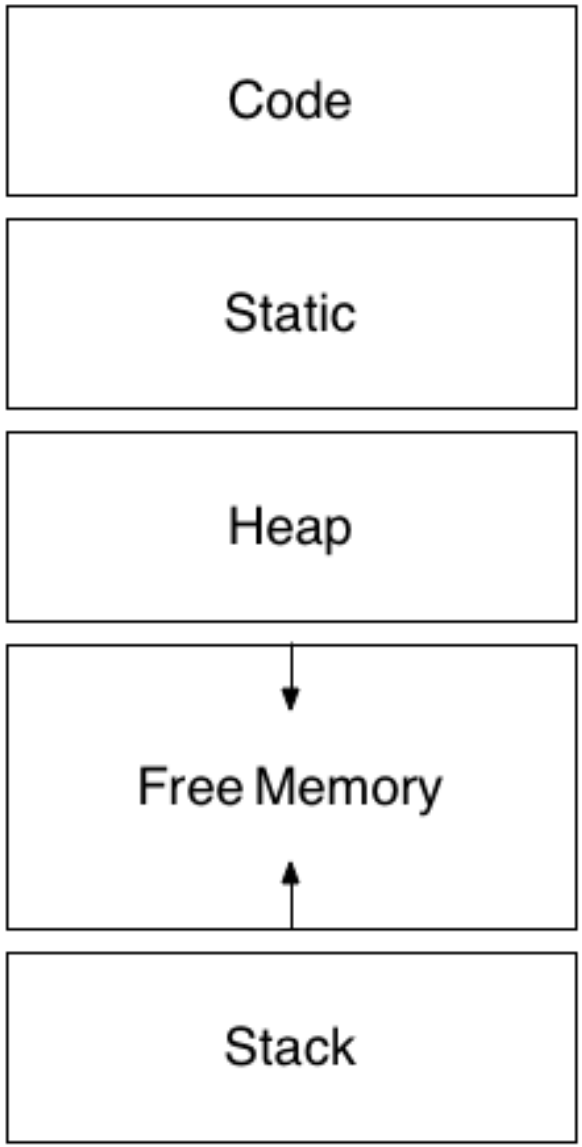
Introduction

- Compilers need to be aware of the run-time environment in which their compiled programs will execute.
- We have already covered a brief overview of activation records.
- We now delve into some additional details ... and
- Discuss a number garbage collection strategies/mechanisms.

Common Memory Layout

- Each executing program has access to its own logical address space.
- The operating system, is responsible for mapping this logical address space into physical addresses, which could be spread throughout memory.
- A program, in this logical address space, consists of data and program areas.
- Run-time storage is exposed by the OS as blocks of contiguous bytes, where a byte is the smallest unit of addressable memory.
- Primitive data types come in handy in determining the amount of storage required.

A typical subdivision of run-time memory

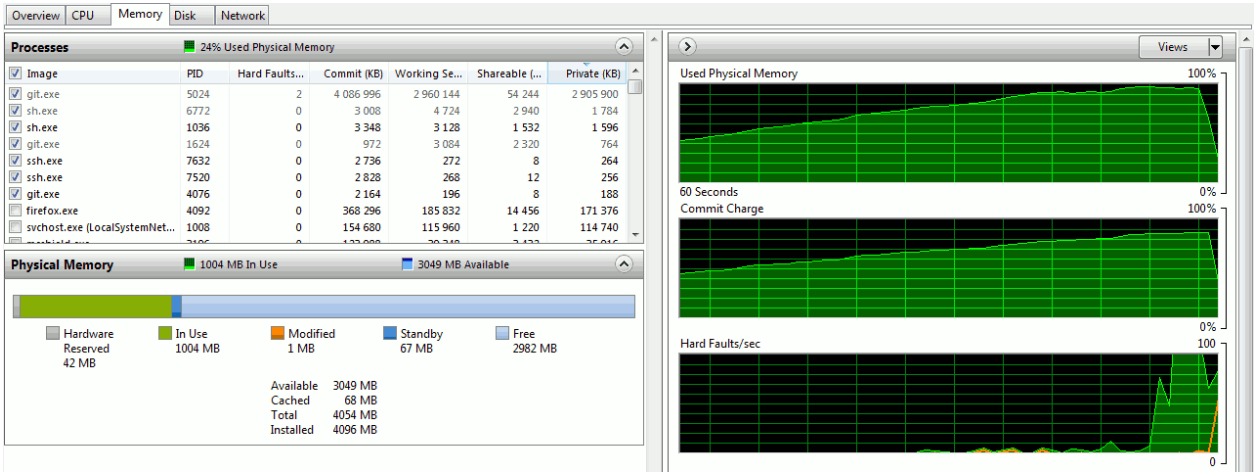


The Stack and the Heap

- The stack and the heap, are at the opposite ends of the remainder of the address space and grow as needed towards each other.
- The **stack** is used to store activation records generated during procedure calls, e.g. names local to a procedure.
- The **heap** is used to manage long-lived data, possibly outliving the call to the procedure that created it.

Static Versus Dynamic Allocation

- If a compiler by looking at the text of a program, i.e. at **compile time**, can determine the size for a particular allocation, then the allocation is done statically, e.g. static variables.
- If size can only be determined at **run time**, then allocation has to be dynamic, e.g. container classes.



The Heap

- Variables which become inaccessible when their procedures exit, are stored in the stack.
- The other variables, whose data lives indefinitely are stored in the heap.
- Their existance therefore, is not tied to the procedure activation that created them.
- Both C++ and Java use the **new** operator to create objects (pointers to them) that may be passed between procedure activations.
- Programs written in C or C++, take care of de-allocation using operators **free** and **delete**, whereas in VM based languages e.g. Java, a GC usually take care of memory de-allocations.

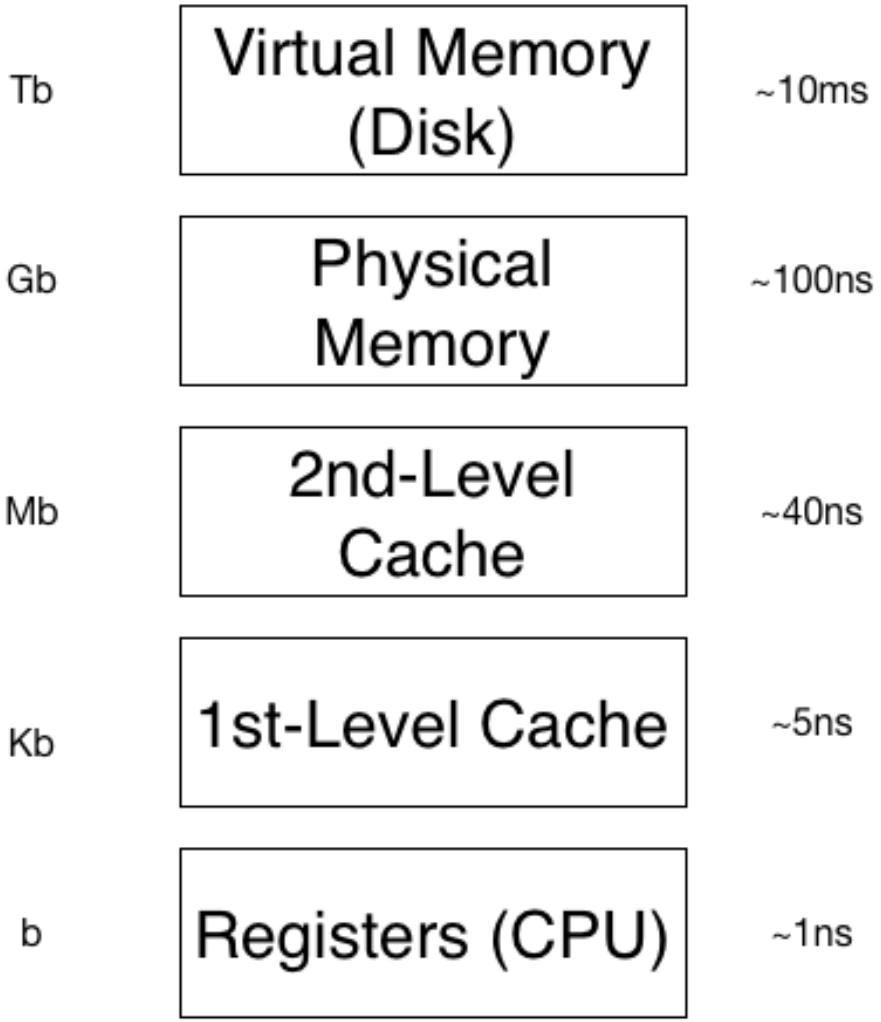
Memory Manager (MM)

- Manages allocations/de-allocations on the heap data structure and keeps track of all the free space at all times.
- It bridges between the application program and the underlying operating system.
- **Allocation**: the MM returns a chunk of **contiguous** heap memory of the requested size. Might itself make requests to the virtual memory managed by the OS if request cannot be handled.
- **De-Allocation**: the MM returns de-allocated space to the pool of free heap space.

Properties of MMs

- Note that i) usually not all allocation requests are of the same size and ii) usually deallocations are not carried out in the same order (reversed) as allocations.
- The MM, therefore must be prepared to service, in any order, allocation and de-allocation requests of any size.
- **Space Efficiency**: Minimize fragmentation in order to minimize total heap space needed by a program.
- **Program Efficiency**: Attention to the placement of objects in memory allows programs to run faster.
- **Low Overhead**: Efficient memory allocations and de-allocations.

Memory Hierarchy



Compiler Optimisations

- Knowledge of how memory is organised is very important for memory managers and compiler optimisations.
- Program efficiency is determined by i) the number of instructions executed and ii) the time taken to execute each instruction.
- In particular, data-intensive programs can benefit significantly, since the time taken to access different parts of the hierarchy differ significantly.
- Register usage is determined by the code a compiler generates. Caches are managed in hardware, whereas virtual memory is managed by the OS.
- Each memory access, incurs a search which is carried out across this memory hierarchy, starting from the lowest level.

Reducing Fragmentation

- At the beginning the heap is one contiguous unit of free space.
- Fragmentation occurs at runtime when the program allocates and de-allocates memory resulting in *holes* in the initial contiguous unit.
- Why is memory fragmentation a problem?

Reducing Fragmentation

- At the beginning the heap is one contiguous unit of free space.
- Fragmentation occurs at runtime when the program allocates and de-allocates memory resulting in *holes* in the initial contiguous unit.
- Why is memory fragmentation a problem?
- A fragmented memory consists of a large number of small, non-contiguous holes making it difficult for the memory manager  to service future allocation requests.

Object Placement

- Fragmentation is reduced by controlling how the memory manager places new objects in the heap. Different strategies are possible:
- **Best-fit**: Allocate the requested memory in the smallest available hole that is large enough. The algorithm tends to spare large holes to satisfy subsequent, larger requests.
- **Next-fit**: Allocate the requested memory in the first hole which it fits. Takes less time but in general inferior to best fit algorithms.

Best-Fit Data Structures

- To improve efficiency, MMs typically organise free space into *bins*, according to their sizes.
- In general, there are many more bins for smaller sizes (16,24,32,...,512 bytes), because there are usually more small objects and less larger sized bins.
- The larger bins, chunks of different sizes are ordered by their size.
- Additional memory can always be requested from the OS (via paging).

Best-Fit Steps

- A typical best-fit algorithm proceeds as follows for a memory allocation request for n bytes:
- If there is a non-empty bin of n sized chunks, take any chunk from that bin,
- If n does not match any of the bins (smaller chunks), we find the bin that is allowed to include chunks of size n. A first-fit or best-fit strategy is applied on this bin in order to choose a sufficiently large chunk of memory. If size of chunk is greater than n, the extra bytes are added to bins of smaller sizes.
- If the required memory chunk is not found, the previous step is applied again on the next bin. Eventually, if required the MM can always request additional pages from the OS.

Coalescing of Free Space

- When an object is de-allocated manually (not GCed), the memory manager must make its chunk free, so it can be allocated again.
- In some cases, the freed chunk can be combined with adjacent free chunks on the heap.
- In advantage, mainly, is that we can always use a large chunk to do the work of small chunks of equal total size, but many small chunks cannot hold one large object, as the combined chunk would.

Problems with Manual Deallocation

- Mainly two!! - **memory leaks** and **dangling pointers**
- *Memory leaks*: or not deleting storage that will never be referenced. In general we can live with them (as long as the machine does not run out of memory).
- *Dangling Pointers*: occurs when storage is reclaimed (de-allocated) and then the program tries to refer to that data.
- Unlike memory leaks, dereferencing a dangling pointer after the freed storage is reallocated almost always creates a program error that is hard to debug!!
- One good reason to use managed languages.

Garbage Collection

- Many programming languages support implicit deallocation of heap objects, i.e. the implementation de-allocates memory objects **automatically** when they are no longer in use.
- This requires some changes to both the (memory) allocator and compiled code generated, i.e. the compiler and run-time must include some mechanism for determining when an object is no longer of interest (dead).
- Garbage collectors adopt these mechanisms to reclaim and recycle this dead space.
- Two main categories - Reference Counting and Trace-Based Collection

Type Safe Languages

- Can GC be applied to all languages? Yes and No !! Not all are ideal candidates.
- Basically, for a GC to work it needs (at runtime) to be able to tell which elements could be used as a pointer to a chunk of allocated memory space, i.e. a language in which the type of any data component can be determined.
- We call these languages, **type safe** languages.
- ML and Java are examples of type safe languages, thus good candidates for garbage collection, whereas C and C++ are bad candidates. Why?

Type Safe Languages

- Can GC be applied to all languages? Yes and No !! Not all are ideal candidates.
- Basically, for a GC to work it needs (at runtime) to be able to tell which elements could be used as a pointer to a chunk of allocated memory space, i.e. a language in which the type of any data component can be determined.
- We call these languages, **type safe** languages.
- ML and Java are examples of type safe languages, thus good candidates for garbage collection, whereas C and C++ are bad candidates. Why?
- Because memory addresses can be manipulated arbitrarily, pointer arithmentic, integers can be cast to pointers, etc.. effectively no memory location can be considered to be inaccessible!!

Reachability

- The term **root set** refers to all the data that can be accessed directly (i.e. without dereferencing a pointer) by a running program. These elements are said to be directly reachable
- For example, in Java the root set consists of all static field members and all variables currently on the stack.
- Reachability is then defined recursively, starting from the root set, i.e. any object with a reference in this reachable set is itself reachable.
- The set of reachable objects changes as a program executes. It grows as new objects are created and shrinks as objects become unreachable.
- For GC purposes, it is important to note that once an object falls out of the reachable set (becomes unreachable), it cannot become reachable again.

Reachability Set Computation

- There are four basic operations that a mutator performs to change the set of reachable objects:
- **Object Allocation**: Performed by MM, returns a reference to the newly allocated chunk of memory. Increments the set of reachable objects.
- **Parameter Passing and Return Values**: References passed between caller and callee (parameters) and value returned by callee. Objects pointed to by these references remain reachable.
- **Reference Assignments**: u = v, where u and v are references. What happens to the objects pointed to *u* and *v*?
- **Procedure Returns**: As a procedure exits, the frame holding its local variables is popped off the stack. If objects are reachable only through these variables, these become unreachable.

Tracking Reachability

- In Summary:
- New objects are introduced through object allocations.
- Parameter passing can propagate reacability.
- Assignments and ends of procedure can terminate reachability.
- As an object becomes unreachable, so may others who were previously reachable through it.
- How do MMs track reachability? Either we catch the event as a reachable object turns unreachable, or the MM periodically locates all the reachable objects and then infers that all the other objects are unreachable.

Reference Counting

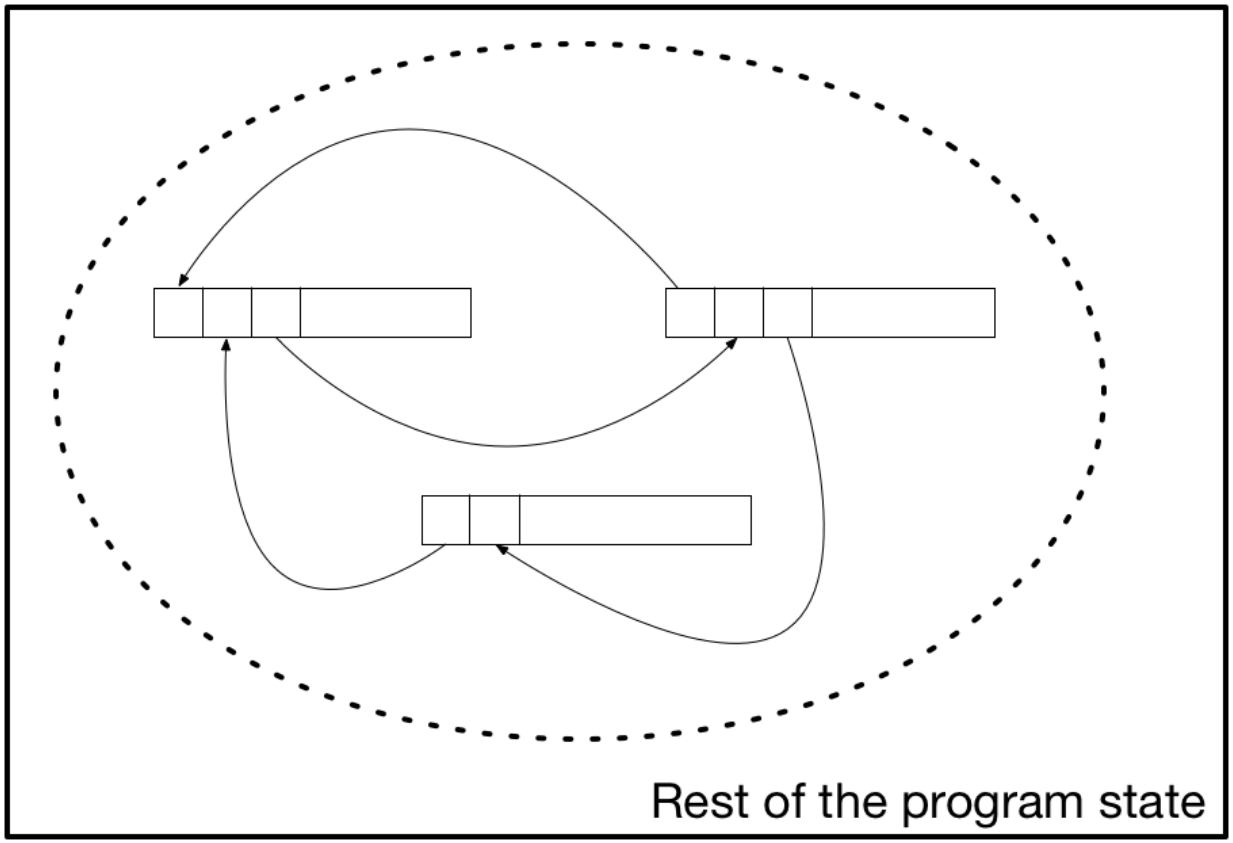
- A technique used when an object's lifetime needs to be determined dynamically. The traditional way of performing incremental garbage collection.
- The technique adds a counter to each heap-allocated object which keeps track of the number of outstanding pointers that refer to the object.
- Whenever a reference to an object is created, the reference counter is incremented and this is removed decremented.
- When this counter goes to zero, there's no way this object can be accessed and therefore its space can be de-allocated.
- What happens with circular data structures!!?
- Resolves dangling-pointer problems but is quite expensive! Why?

Reference Counting GC

- Reference counts can be maintained as follows:
- *Object Allocation*: The reference count of the new object is set to 1.
- *Parameter Passing*: The reference count of each object passed into a procedure is incremented.
- *Reference Assignment u=v*: The reference count of the object refered to by v going up by 1 and the reference count of the object refered to by u goes down by one.
- *Procedure Returns*: As a prodecure returns all the references held by the local variables of that procedure activation record must also be decremented.
- *Transitive Loss of Reacability*: Whenever the reference count of an object becomes zero, we must also decrement the count of each object pointed to by a reference within the object.

Disadvantages of RC Collectors

- Cyclic Data Structures, e.g. data structures which point back to their parents, could lead to memory leaks.



Disadvantages of RC Collectors

- Performance - Additional overheads are introduced with each reference assignment, procedure entries and exists.
- The overheads incurred increase proportional to the amount of computation in the program, and not just the number of objects in the system, which could be significant.
- The C++ Boost library provides templated smart pointers which essentially use a reference counting mechanism.

Trace-based Batch Collectors

- Instead of collecting garbage as it is created (reference counting), trace-based collectors run periodically (or when invoked) to find unreachable objects and deallocate their space.
- Typically, batch collectors consider deallocation when the free-space pool has been exhausted (or close to).
- These collectors, pause the execution of the program, examine the pool of allocated memory to discover unused objects, and reclaims their space.
- The traditional way of performing batch-oriented garbage collection is sometimes referred to as stop-the-world collector.
- As with ref counting, algorithms exist that perform collection incrementally to amortize the cost over longer periods of execution time.

Phases in a Trace-based Collector

- It's goal is to find all the unreachable objects, and put them on the list of free space.
- Phase 1: Discover the set of objects that can be reached from pointers stored in the program variables and compiler generated temporaries. The collector assumes that any object reachable in this manner is live. The rest are dead!!
- —
- Phase 2: Deallocate/recycle dead objects. Two common techniques used are
  - Mark-and-sweep or mark-and-compact,
  - Copying collectors.

Marking in a Mark-and-Sweep GC

- A marking algorithm is used, which assigns a *mark bit* to each object in the heap. The initial step clears these bits and builds a worklist containing all the pointers stored in registers and in variables accessible (root set).
- The second step of the algorithm traverses these pointers and marks every object that is reachable.

Algorithm 1 A simple Marking Algorithm

```
1: Clear all marks
2: Unscannedlist ← pointer values from root set
3: while Unscannedlist ≠ 0 do
4:     remove p from the Unscannedlist
5:     if (p → object is unmarked)
6:         mark p ← object
7:         add pointers from p → object to Unscannedlist
8: end while
```

Mark-and-Sweep Algorithm

- The second is the sweeping phase.

Algorithm 2 A Mark and Sweep Garbage Collector

```
1: Free = 0;
2: for each chunk of memory o in the heap do
3:     if (o is unreached, i.e., its reached-bit is 0) add o to Free;
4:     else set the mark-bit of o to 0;
5: end for
```

Basic Abstraction for Trace-Based Collectors

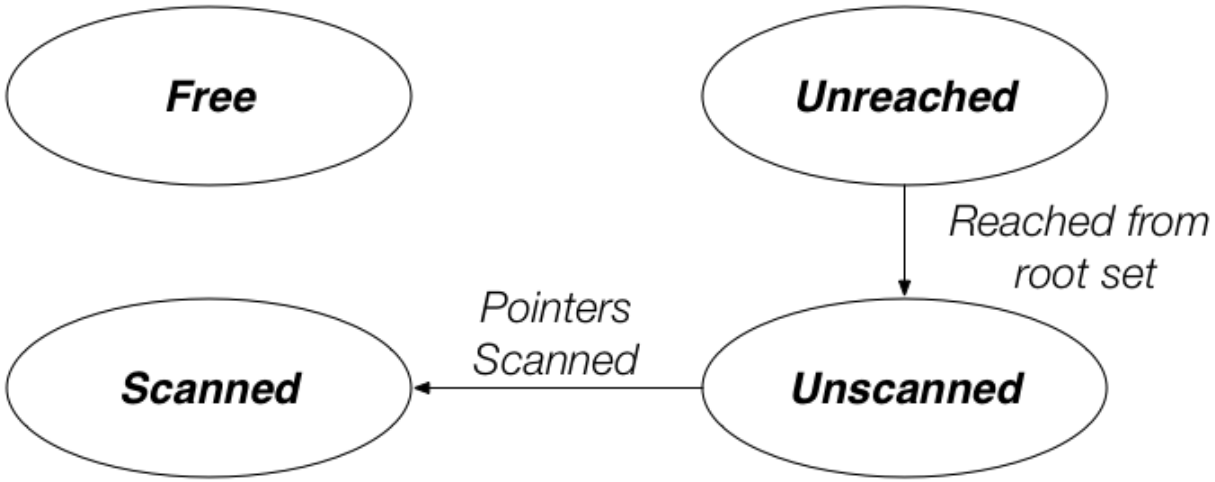
- All trace-based algorithms compute the set of reachable objects and then take the complement of this set.
- From the memory point-of-view, how is its state changing during Garbage Collection?
- A number of terms describe these states, namely:
  - *Free*: Ready for allocation.
  - *Unreached*: Chunks are presumed unreachable unless proved otherwise.
  - *Unscanned*: If a chunk is reachable, it becomes unscanned. i.e. its pointers have not been scanned yet.
  - *Scanned*: When the scan of an object is complete, its state is set to scanned.

From Unscanned to Scanned - To Scan an Object

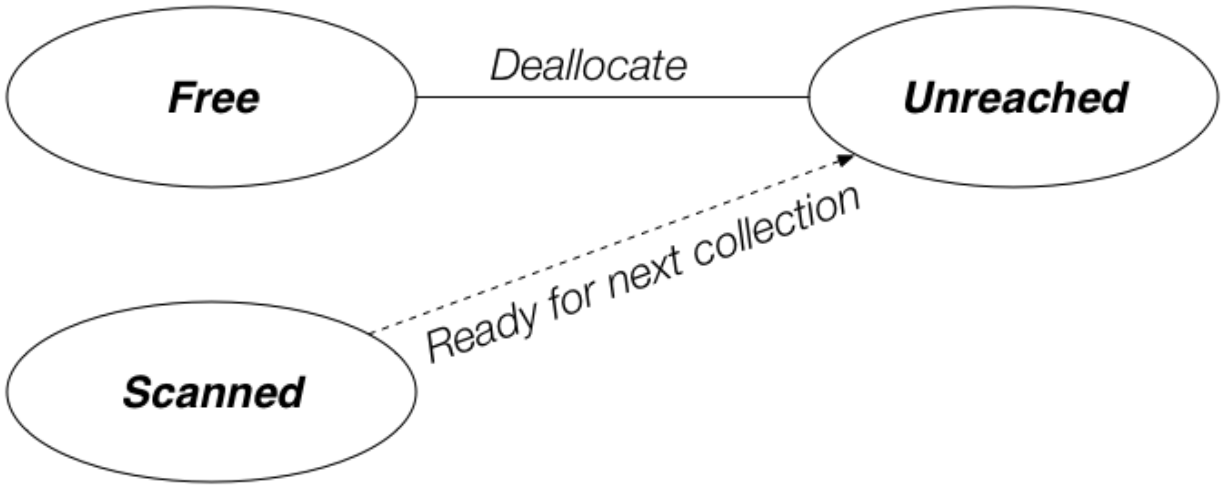
- Chunks known to be reachable are either in state unscanned or *scanned*.
- Every *unscanned* object will eventually be scanned and transition to the *scanned* state.
- To **scan** an object, all pointers (references) in the object are followed. If this referenced object is in state *unreached*, that objects state transitions to *unscanned*
- When the scan of an object is complete, it transitions to the *scanned* state. Note that a scanned object can only contain references to other *scanned* or *unscanned* objects, and never to *unreached* objects.
- The computation of reachability of complete when no objects are left in the *unscanned* state. Objects still in the *unreached* state are reclaimed and transition to the *free* state, while the rest transition to the *unreached* state and get ready for another GC invocation.

States of Memory Chunks - Diagram

States of Memory Chunks - Diagram



States of Memory Chunks - Diagram



Baker's Mark-and-Sweep Optimisation

- The basic mark-and-sweep algorithm is expensive since it has to examine the entire heap to determine the reachable set.
- Basically the algorithm does not know what allocations were carried out. Baker's mark-and-sweep improves on the basic algorithm by simply keeping track of allocated objects.
- The set of unreachable objects, is then computed by taking the set difference of the allocated objects and the reached objects.
- The input consisits of a root set of objects, a heap, a free list *Free*, and a list of allocated objects, refered to as *Unreached*
- Note that this algorithm simply maps the memory states just mentioned, to four lists storing objects in these states.

Baker's Mark-and-Sweep collector

Algorithm 3 A simple Marking Algorithm

```
1:  Scannedlist = 0;
2:  Unscannedlist = set of objects referenced in the root set;
3:  while Unscannedlist ≠ 0 do
4:      move object o from Unscannedlist to Scannedlist
5:      for each object o' referenced in o do
6:          if (o' is in Unreachedlist)
7:          move o' from Unreachedlist to Unscannedlist;
8:      end for
9:  end while
10: Freelist = Freelist ∪ Unreachedlist;
11: Unreachedlist = Scannedlist
```

Mark-and-Compact Garbage Collectors

- In mark-and-sweep, it has been assumed that chunks returning to the *Freelist*, remain as there were before deallocation. However, as we have discussed earlier, it is usually the case that adjacent free chunks are combined into larger chunks in order to reduce fragmentation.
- A different approach is taken by Mark-and-Compact collectors (a.k.a relocating collectors) which move reachable objects around in the heap to eliminate memory fragmentation.
- The heuristic used is that (usually) the space occupied by reachable objects is much smaller that the free space.
- After identifying all the unreachable objects (holes) and freeing them individually, a M-and-C collector relocates all the reachable objects to one end of the heap and free the rest.

Mark-and-Compact Variations

- Having all reachable objects in contiguous space imporoves temporal and spatial locality of a program.
- It also simplifies the data structure used for maintaining free space; instead of using *Freelist* the collector simply needs a pointer to the beginning of the one free block.
- Relocating collectors mainly vary in whether they relocate in place or reserve space ahead of time for the relocation:
  - The traditional Mark-and-Compact, compacts objects in place thus reducing memory usage.
  - A *copying collector* moves objects from one region of memory to another. Reachable objects are moved as they are discovered.

Mark-And-Compact Algortihm Overview

- 1) A marking phase similar to the mark-and-sweep algorithms described.
- 2) The algorithm scans the allocated section of the heap and computes a new address for each of the reachable objects. New addresses are assigned from the low end of the heap. The new address for each object is stored in a structure called *NewLocation*
- 3) The algorithm copies objects to their new locations, updating all references in the objects to point to the corresponding new locations.
- The output of the algorithm is a new value for the pointer *free* marking the beginning to the free chunk.

Mark-And-Compact Psuedo-code

Algorithm 4 Mark-and-Compact Collector - Mark

```
1: Unscannedlist = set of objects referenced in the root set;
2: while Unscannedlist ≠ 0 do
3:     remove object o from Unscannedlist
4:     for each object o' referenced in o do
5:         if (o' is unreached)
6:         mark o' as reached;
7:         put o' on unscannedlist;
8:     end for
9: end while
```

Mark-And-Compact Psuedo-code

Algorithm 5 Mark-and-Compact Collector - Compute New Locations

```
1: free = starting location of heap storage;
2: for each chunk of memory o in the heap, from the low end do
3:     if (o is reached)
4:     NewLocation(o) = free;
5:     free = free + sizeof(o);
6: end for
```

Mark-And-Compact Psuedo-code

Algorithm 6 Mark-and-Compact Collector - Retarget References

```
1: for each chunk of memory o in the heap, from the low end do
2:     if (o is reached)
3:     for each reference o.r in o do
4:         o.r = NewLocation(o.r);
5:         copy o to NewLocation(o);
6:     end for
7: end for
8: for each reference r in the root set do
9:     r = NewLocation(r);
10: end for
```

Copying Collectors - C.J. Cheney

- A copying collector, reserves ahead of time, space to which the objects can move.
- The biggest advantage is that this breaks the dependency between tracing and finding free space.
- Memory is partitioned into two halfspaces, A and B.
- New objects are created in one of these halfspaces, until it fills up, at which point the mutator (MemManager) is stopped and the garbage collector copies the reachable objects to the other space.
- When GC'tion completes, the roles of the halfspaces are reversed.

A Copying Collector Psuedo Code

Algorithm 7 Copying GC

```
1:  for all objects o in From space do
2:      NewLocation(o) = NULL;
3:  end for
4:  unscanned = free = starting address of To space;
5:  for each reference r in the root set do
6:      replace r with LookupNewLocation(r);
7:  end for
8:  while unscanned ≠ free do
9:      o = object at location unscanned;
10:     for each reference o.r referenced in o do
11:         o.r = LookupNewLocation(o.r);
12:     end for
13:     unscanned = unscanned + sizeof(o);
14: end while
```

The LookupNewLocation Function

- Thus function takes an object o and finds a new location for it in the *To* halfspace if it has no location there yet. All new locations are recorded in a structure (e.g. Hashmap) *NewLocation*

Algorithm 8 Copying GC LookupNewLocation()

```
1: if (NewLocation(o) = NULL); {
2:       NewLocation(o) = free;
3:       free = free + sizeof(o);
4:       copy o to NewLocation(o);
5: }
6: Return NewLocation(o);
```

Costs of Garbage Collection

- Basic Mark-and-Sweep: Proportional to the number of chunks in the heap.
- Baker's Mark-and-Sweep: Proportional to the number of reached objects.
- Basic Mark-and-Compact: Proportional to the number of chunks in the heap plus the total size of the reachable objects.
- Cheney's Copying Collector: Proportional to the total size of the reached objects.

More Advanced Topics

- We have described stop-the-world garbage collection techniques. However these are not always ideal due to the possibility of introducing long pauses into the execution of programs. This cost can be amortized over the execution time of the program.
- Short-Pause Garbage Collection
  - Incremental Garbage Collection
  - Incremental Reachability Analysis
- Generational Garbage Collection
- Parallel and Concurrent Garbage Collection
- Unsafe Languages