“A Garbage Collector For the C Programming Language “

V.R.Siddhartha Engineering College

Vijayawada

Presented by

CH.Sreekanth

¾ B-TECH (IT)

Email:shri136@gmail.com

CH.Sekhar Babu

¾ B-TECH (IT)

Email:sekhar_chandra64@yahoo.com

ABSTRACT

Garbage collection absolves the developer from tracking memory usage and knowing when to free memory. If unreferenced locations are not collected, they pollute the memory and may lead to exhaustion of available memory during the execution of the program. Garbage collection (GC) is an important part of many modern language implementations. The benefits of garbage collection in comparison with manual memory management techniques are well known to programmers who have used high-level languages like Java and C#. Software professionals estimate that the programming effort required to manually perform dynamic memory management is approximately 40% of the total cost of developing a large software system. Unfortunately, these benefits have not traditionally been available to developers in 'C' programming language because of the lack of support for the GC feature.

We have developed a library that builds in this feature, into the 'C' programming language and thus improving its functionality. We have in our library a set of routines that embed the feature of automatic dynamic memory management in to the 'C' programming language. These routines defined in our library form, the programmer's Interface to our garbage collector. The interface is simple and the syntax is very much similar to that of the one already available in the standard library (stdlib.h) with very little modifications. In addition to the routines that support GC we also have routines that support Finalizers amongst others.

INTRODUCTION

C is and has always been one of the most widely and popularly used programming languages in the world. We, as a part of our B.E Degree course have done a lot of programming in C and have always used the built in functions defined in the standard library like "malloc", "calloc" etc. to allocate memory and when the allocated memory is not needed any more we use the built in function "free" to free up the allocated memory. What we have attempted to improve the functionality of the 'C' programming language by developing a library that supports garbage collection. We were inspired by the presence of the GC features in JAVA and proceeded to develop one for C. So now, the 'C' programmer is relived of the task of keeping track of what memory locations are currently being referenced and what are not. If the programmer uses our library for allocating memory dynamically, we will make sure (conservatively) that the allocated memory is automatically freed (i.e.) garbage is collected when it is no longer referenced.

Myths about GC

- GC is necessarily slower than manual memory management.
- GC will necessarily make my program pause.
- Manual memory management won't cause pauses.
- GC is incompatible with C.

Facts about GC

- Most allocated objects are dynamically referenced by a very small number of Pointers.
- The most important small number is ONE.
- Most allocated objects have short lifetimes.
- Allocation patterns (size distributions, lifetime distributions) are bursty, not uniform.
- Optimal" strategies can fail miserably.

Garbage Collector and Finalizers

A finalizer is some code that runs when an object is about to be freed. One of the main uses of destructors is to perform clean up activities. A couple of typical uses are listed below as follows,

One use is for objects that have a state outside the program itself. The canonical example is an object that refers to a file. When a file object becomes eligible for reclamation, the garbage collector needs to ensure that buffers are flushed, the file is closed and resources associated with the file are returned to the operating system.

Another use is where a program wants to keep a list of objects that are referenced elsewhere. The program may want to know what objects are in existence for, say accounting purposes but does not want the mechanism of accounting to prevent objects from otherwise being freed. Garbage Collection Algorithms

In our collector the programmer can specify the function (finalizer) that he wishes be called when the object is to be freed to perform clean up activities.

GARBAGE COLLETORS

Any garbage collection algorithm must ensure two basic things. First it must detect garbage objects. Second it must reclaim the heap space used by the garbage objects and make the space available again to the program.

Garbage detection is ordinarily accomplished by defining a set of roots and determining *reachability* from the roots. An object is said to be reachable if there is some path of reference from the roots by which the executing program can access the object. The roots are always accessible to the program. Any object that is reachable from the roots is considered to be *live.* Objects that are not reachable are considered garbage as they can no longer affect the future course of program execution.

Some garbage collectors, however, may choose not to distinguish between genuine object references and look-alikes. Such garbage collectors are called *conservative* because they may not always free every unreferenced object. Sometimes a garbage object will be wrongly considered to be live by a conservative collector, because an object reference look alike referred to it. Conservative collectors trade-off an increase in garbage collection speed for occasionally not freeing some actual garbage.

Two basic approaches in distinguishing live objects from garbage are *reference Counting* and *tracing*. Reference counting garbage collectors distinguish live objects from garbage objects by keeping a count for each object on the heap. The count keeps track of the number of references to that object. Tracing garbage collectors actually trace out the graph of references starting with the root nodes. Objects that are encountered during the trace are marked in some way. After the trace is complete, unmarked objects are known to be unreachable and can be garbage collected

ARCHITECTURE AND DESIGN

Our garbage collection algorithm is invoked when any one of the following conditions are met. The conditions are as follows,

a) Whenever the number of bytes that have been allocated by our allocation routine exceeds a predefined limit. That is why we call our collector to be periodic and the programmer can set this period if he wishes to. By default this value has been set to 640K bytes of memory.

b) Whenever the allocation routine is unable to allocate the memory as such. That is if malloc returns NULL. In this case we call our collector to collect the garbage and then try to re-satisfy the request for memory.

Our GC Library provides the following routines and they act as the programmer's interface to the collector.

a)void * GC_malloc(size_t gc_size)

Input: gc_size

Output: Starting address of the memory location that has just been allocated if successful and NULL if allocation fails.

Description: This function allocates a memory location of size gc_size and returns the address of that memory location if successful in allocation. If the requested memory cannot be allocated even after Garbage collection then this routine returns null.

b)Void*GC_malloc_uncollectable(size_t gc_size)

Input: gc_size

Output: Starting address of the memory location that has just been allocated if successful and NULL if allocation fails.

Description: This function allocates a memory location of size gc_size and returns the address of that memory location if successful in allocation. If the requested memory cannot be allocated even after Garbage collection then this routine returns NULL. The specialty of this function is the memory allocated by GC_malloc_uncollectable will not be freed during the garbage collection

c) void * GC_malloc_finalizer (size_t gc_size , void (* gc_finalizer)())

Input: gc_size, gc_finalizer

Output: Starting address of the memory location that has just been allocated if successful and NULL if allocation fails.

Description: This function allocates a memory location of size gc_size and returns the address of that memory location if successful in allocation. If the requested memory cannot be allocated even after Garbage collection then this routine returns NULL. The specialty of this function is the memory allocated by GC_malloc_finalizer will be freed during the garbage collection and during the freeing of this memory location the function gc_finalizer will get executed

d) void GC_collect_garbage()

Input: None

Output: None

Description: This function locates the garbage mark them and then it collects the same. The user can periodically call this function to collect the garbage. This may improve the performance.

e) void GC_free (void * gc_waste)

Input: The address of the memory location to be freed.

Output: None

Description: This function is responsible for freeing the memory location gc_waste and add them to the free list. The memory allocated using the function GC_malloc_uncollectable can be freed by using this function.

f) void GC_set_gc_rate (size_t max_heap_size)

Input: It takes the memory size as argument.

Output: None

Description: This function is responsible for setting the threshold value for allocated memory. If the memory allocated dynamically by the user exceeds the threshold, the garbage collection routine will be called automatically.

IMPLEMENTATION METHODOLOGY

The Two-Phase Abstraction:

Garbage collection automatically reclaims the space occupied by data objects that the running program can never access again.

Such data objects are referred to as garbage. The basic functioning of our garbage collector consists, abstractly speaking of two phases.

- Distinguishing the live objects from the garbage or *garbage detection*.
- Reclaiming the garbage objects storage so that the running program can use it or *Garbage collection.*

Algorithm – An Overview

1) User asks for allocation of memory.

2) We try to allocate the requested amount of memory.

3) If memory request is satisfied and memory so far allocated < GC Limit

```
Begin
        Do book-keeping activity;
        Return the address that corresponds to
        the allocated memory region.
End
```

4) If memory request is satisfied and memory so far allocated > GC Limit

```
Begin
        Call the GC routine;
        Do book-keeping activity;
        Return the address that corresponds to
        the allocated memory region;
End
```

5) If memory request is not satisfied

```
Begin
        Call the GC routine;
        Try to reallocate;
        If (allocation satisfied)
                Return address of the allocated
                object
        Else
                Return ERROR;
        End if
End
```

Detailed Algorithm adopted by our Garbage Collector Steps

1) The user requests for allocation using the function "GC_malloc" or any of its Variants.

2) We try to satisfy his request by allocation an object of the size requested by the user from the heap.

3) If the allocation in step 2 succeeds then we return the address of the allocated object after doing book-keeping activity

4) If during the process of allocation in step 2 we were unable to allocate or if the allocation so far done has exceeded a threshold limit (640kb by default), then we call our GC routine that tries to collect unused memory (i.e. garbage).

5) Our GC Routine works as follows.
From the addresses that have been allocated (these are available from the bookkeeping entries) search which of the objects pointed to by these addresses are live and which are not. This can be found by

i. First making a search that will tell us help us to find the set of root pointers.

ii. The book keeping list gives us the information about what to search? And the root pointers will give us the information about where to search?.

iii. Having got the information about what to search? and where to search?, our algorithm gets in to the mark phase first for finding and marking the garbage and then in to the sweep phase for collecting the garbage.

Mark Phase

If an object is found to be alive then mark it. This is done by setting a field in the book-keeping node corresponding to this object.

Sweep Phase

Scan all the objects. If an object is marked then unmark it. If an object is not marked then free it as it has been declared as garbage by the mark phase of our collector.

6) **Our algorithm**

.

Exits printing the message " Your request cannot be satisfied even after garbage collection", if allocation fails even after trying to garbage collect.

.

Returns the pointer to the allocated object of the size requested by the user if allocation was successful

CONCLUSION AND FUTURE IMPROVEMENTS

- Our garbage collection routine currently pauses the program when it runs. What can be done is to make the garbage collection routine to run in a separate thread in parallel in such a way that the program does not stop when GC routine/collector runs.
- . We do not expect the addresses of the memory locations allocated by our garbage collection routine to be stored in the CPU registers because we are yet to implement the collector module which searches for the availability of pointers to live objects in the CPU registers.
- . Our garbage collection routine currently uses a sequential search in the mark and sweep phases to check for the liveliness of the object. In order to improve the performance a hashing function can be implemented replacing the sequential search.
- . Our garbage collection routine is currently platform dependent. Modifications can be made to the code in such a way that the collector can be used in a variety of platforms.
- . A comparative study can be done between our collector and some other existing garbage collectors for C based on some standard criteria so that modifications (if any needed) can be done to our collector to identify any shortcomings and pitfalls and improve its efficiency if so required

BIBLIOGRAPHY

Books:

- Advanced Programming In The Unix Environment by W. Richard Stevens.
- . The Design Of The Unix Operating System by Maurice J. Bach.
- . The Unix Programming Environment by Brian W. Kernighan and Rob Pike.
- . The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie.
- . Inside The Java 2 Virtual Machine – Garbage Collection by Bill Venners.

Papers:

- . Uniprocessor Garbage Collection Techniques by Paul R. Wilson.
- . Garbage Collection: Automatic Memory Management In The Microsoft's .NET Framework by Jeffrey Richter.
- . Incremental Mature Garbage Collection Using The Train Algorithm by Jacob Seligmann and Steffen Grarup.

Websites:

- www.hpl.hp.com/personal/Hans_Boehm/gc/
- . www.citeseer.com
- . www.memorymanagement.com
- . www.cprogramming.com