Uniprocessor Garbage Collection Techniques

Presented by:

Shiri Dori

Shai Erera

Outline

- What is Garbage Collection
- Basic Garbage Collection Techniques
- Advanced Techniques
  - Incremental Garbage Collection
  - Generational Garbage Collection
- Language-Related Features

Garbage Collection

- Garbage Collection (**GC**) is the automatic storage reclamation of computer storage
- The GC function is to find data objects that are no longer in use and make their space available by the running program

So Why Garbage Collection?

- A software routine operating on a data structure should not have to depend what other routines may be operating on the same structure
- If the process does not free used memory, the unused space is accumulated until the process terminates or swap space is exhausted

Explicit Storage Management Hazards

- Programming errors may lead to errors in the storage management:
  - May reclaim space earlier than necessary
  - May not reclaim space at all, causing memory leaks
- These errors are particularly dangerous since they tend to show up after delivery
- Many programmers allocate several objects statically, to avoid allocation on the heap and reclaiming them at a later stage

Explicit Storage Management Hazards

- In many large systems, garbage collection is partially implemented in the system's objects
  - Garbage Collection is not supported by the programming language
  - Leads to buggy, partial Garbage Collectors which are not useable by other applications
- The purpose of GC is to address these issues

GC Complexity

- Garbage Collection is sometimes considered cheaper than explicit deallocation
- A good Garbage Collector slows a program down by a factor of 10 percent
- Although it seems a lot, it is only a small price to pay for:
  - Convenience
  - Development time
  - Reliability

Garbage Collection – The Two-Phase Abstraction

- The basic functioning of a garbage collector consists, abstractly speaking, of two parts:
  - Distinguishing the live objects from the garbage in some way (*garbage detection*)
  - Reclaiming the garbage objects' storage, so that the running program can use it (*garbage reclamation*)
- In practice these two phases may be interleaved

Basic Garbage Collection Techniques

- The first part of a Garbage Collector, distinguishing live objects from garbage, can be done in two ways:
  - *Reference Counting*
  - *Tracing*
- There are several varieties of tracing collection which will be discussed later

Reference Counting

- Each object has an associated count of the references (pointers) to it
- Each time a reference to the object is created, its reference count is increased by one and vice-versa
- When the reference count reaches 0, the object's space may be reclaimed

Example

HEAP SPACE

ROOT SET

1 1 2 1 1 1 2 1 1

Reference Counting – Cont.

- When an object is reclaimed, its pointer fields are examined and every object it points to has its reference count decremented
- Reclaiming one object may therefore lead to a series of object reclamations
- There are two major problems with reference counting

The Cycles Problem

- Reference Counting fails to reclaim circular structures
- Originates from the definition of *garbage*
- Circular structures are not rare in modern programs:
  - Trees
  - Cyclic data structures
- The solution is up to the programmer

Reference Counting

HEAP SPACE

ROOT SET

1 1 1 1 1 1 2 1 1

The Efficiency Problem

- When a pointer is created or destroyed, its reference count must be adjusted
- Short-lived stack variables can incur a great deal of overhead in a simple reference counting scheme
- In these cases, reference counts are incremented and then decremented back very soon

Deferred Reference Counting

- Much of this cost can be optimized away by special treatment of local variables
- Reference from local variables are not included in this bookkeeping
- However, we cannot ignore pointers from the stack completely
- Therefore the stack is scanned before object reclamation and only if a pointer's reference count is still 0, it is reclaimed

Reference Counting - Recap

- While reference counting is out of vogue for high-performance applications,
- It is quite common in applications where acyclic data structures are used
- Most file systems use reference counting to manage files and/or disk blocks
- Very simple scheme

Mark-Sweep Collection

- Distinguishing *live* object from *garbage*
  - Done by tracing – starting at the root set and usually traversing the graph of pointers relationships
  - The reached objects are *marked*
- Reclaiming the *garbage*
  - Once all *live* objects are marked, memory is exhaustively examined to find all of the unmarked (garbage) objects and reclaim their space

Mark-Sweep Collection

- There are three major problems with traditional mark-sweep garbage collectors:
  - It is difficult to handle objects of varying sizes without fragmentation of the available memory
  - The cost of the collection is proportional to the size of the heap, including live and garbage objects
  - Locality of reference

Mark-Compact Collection

- Mark-Compact collectors remedy the fragmentation and allocation problems of mark-sweep collectors
- The collector traverses the pointers graph and copy every *live* object after the previous one
- This results in one big contiguous space which contains *live* objects and another which is considered free space

Mark-Compact Collection

- *Garbage* objects are “squeezed” to the end of the memory
- The process requires several passes over the memory:
  - One to computes the new location for objects
  - Subsequent passes update pointers and actually move the objects
- The algorithm can be significantly slower than Mark-Sweep Collection

http://www.artima.com/insidejvm/applets/HeapOfFish.html

Copying Garbage Collection

- Like Mark-Compact, the algorithm moves all of the *live* objects into one area, and the rest of the heap becomes available
- There are several schemes of copying garbage collection, one of which is the “Stop-and-Copy” garbage collection
- In this scheme the heap is divided into two contiguous *semispaces*. During normal program execution, only one of them is in use

Stop-and-Copy Collector

- Memory is allocated linearly upward through the “current” semispace
- When the running program demands an allocation that will not fit in the unused area,
- The program is stopped and the copying garbage collector is called to reclaim space

Copying Garbage Collection

ROOT SET

FROMSPACE

TOSPACE

Copying Garbage Collection

ROOT SET

FROMSPACE

TOSPACE

Copying Garbage Collection

- Can be made arbitrarily efficient if sufficient memory is available
- The work done in each collection is proportional to the amount of live data
- To decrease the frequency of garbage collection, simply allocate larger semispaces
- Impractical if there is not enough RAM and paging occurs

Choosing Among Basic Tracing Techniques

- A common criterion for high-performance garbage collection is that the cost of collecting objects be comparable, on average, to the cost of allocating objects
- While current copying collectors appear to be more efficient than mark-sweep collectors, the difference is not high for state-of-the art implementations

Choosing Among Basic Tracing Techniques

- When the overall memory capacity is small, reference counting collectors are more attractive

- Simple Garbage Collection Techniques:
  - Too much space, too much time

Advanced Approaches

- Two advanced yet conflicting approaches:
- Incremental Tracing
  - Suits Real-Time environments, where time matters
  - Works in parallel to the program
- Generational Collection
  - Collects better, based on age of objects
  - Hides time from user, but not good for Real-Time

Incremental Tracing

- Real-Time Systems have time constraints
- The garbage collector works in parallel to the program, as a concurrent process
- Must have a way to keep track of changes that the program makes during the collection cycle
  - While the collector “isn’t looking”…

Consistency as Coherence

- Both Program and Garbage Collector access the data structure
- This is akin to coherence among processes:
  - Incremental Mark-Sweep
    - Multiple Read, Single Write (only by the program)
  - Copying Collectors - harder!
    - Multiple Read, Multiple Write (program and GC)

- Solution – views don’t have to be identical

Conservatism

- As long as the different views don't harm execution, a garbage collector can be **conservative**
  - Might view unreachable objects as reachable
  - But not the opposite – that causes errors
- This "Floating Garbage" is guaranteed to be collected during the *next* cycle
  - Unfortunate, but essential
  - Allows cheaper coordination

Tricolor Marking Abstraction

- Collection can be viewed as traversing a graph of reachable objects and coloring them
  - White – haven't reached it yet
  - Gray – reached it, but not traversed all edges originating from it (i.e. not reached all sons)
  - Black – reached it and all its edges (sons)
- "A wavefront of gray objects, which separates the white from the black"
- When finished tracing, white objects are unreachable and can be reclaimed

Wavefront Advancement

A

B C

D

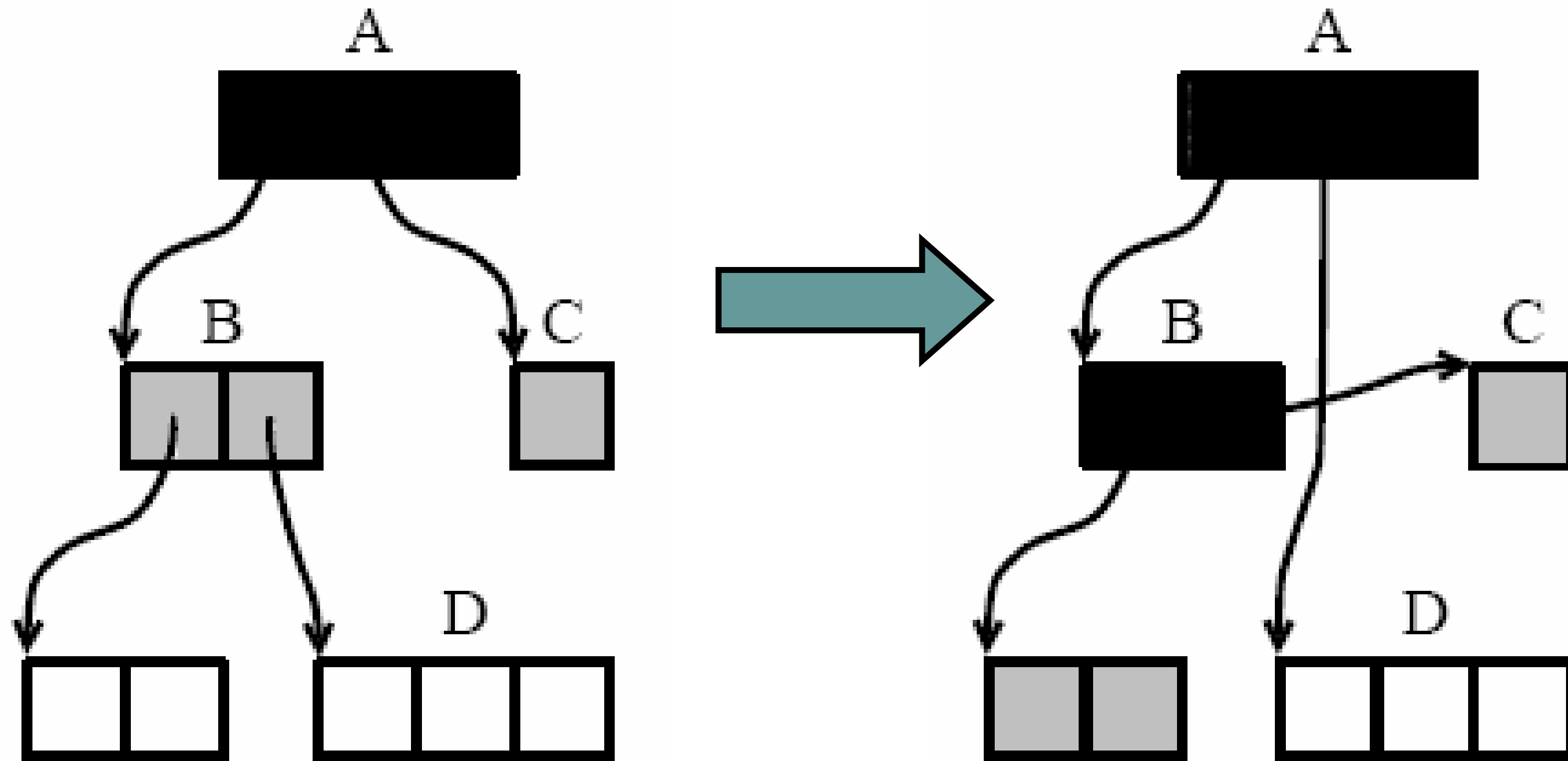
A

B C

D

Violation of Coloring Invariant

- Suppose the program changed the pointers:



Tricolor Marking Abstraction
– Changes by the Program

- If the program creates a pointer from black to white, the GC has to be aware of this
- This can be done in several ways:
  - Read Barrier – whenever reading a white object, it becomes gray
    - This is very expensive
  - Write Barrier – whenever a pointer is written, the write is recorded or acted upon

Quick Overview – Incremental Algorithms

- Several incremental algorithms exist:
  - Snapshot-at-Beginning
  - Baker's Read Barrier (Copying, non-copying)
  - Replication Copying
  - Incremental Update (Dijkstra's, Steele's)

- They differ in how close they are to the real view of the data-structure

Incremental Algorithms – Snapshot-at-Beginning

- Creates a *copy-on-write* virtual copy of the graph; uses write barrier to maintain it
- Traverses graph **as it was** at start of cycle
- Ignores objects that were freed in the meantime – lots of Floating Garbage
- Therefore, new objects treated specially
  - Allocated black, to avoid a useless traversal
- Extra Conservative

Incremental Algorithms – Incremental Update (Steele)

- Tries (heuristically) to retain only objects live at the end of the collection
- Uses Write Barrier – if new pointer installed in black object, it is grayed and traversed again to find any live objects
- New objects are allocated white
  - They might die before ever being collected!
  - Risky…
- Non-Conservative – willing to recompute

Conservatism and Coherence

- There is a big tradeoff between:
  - Coordination costs – to be most up-to-date, and
  - Conservatism wastes – lots of floating garbage
- The less conservative an algorithm, the higher cost of being coordinated
- E.g. Dijkstra’s version of Incremental Update
  - In a stack with many pushes and pops
    - The objects will be reclaimed in Steele’s, by undoing
    - Not in Dijkstra’s, but its cost will be lower

Opportunistic Traversal

- It matters very much in what order the objects are traversed!
- An object will not be reached if all paths to it are broken before the collector touches them
- So, traversing the graph in a smart way will impact performance (not just BFS, DFS)
    - E.g. delay traversing rapidly-changing objects

Really Real-Time

- Incremental is well-suited for Real-Time
- GC must impose only small delays on the program
  - Although the scale of “small delay” may change
  - Can take advantage of mixed time-scales
- Must consider Worst-Case, not Average!
- GC wastes both time and space necessary for the Real-Time application

Real-Time Garbage Collection – Problems and Solutions

- Root Set is usually treated at once
  - Large root set means large initial operation
  - Can be solved by treating some (or all) variables as being on the heap
  - **But** – wastes time on a bigger read/write barrier

Real-Time Garbage Collection – Problems and Solutions

- Root Set is usually treated at once

- Must guarantee progress of freeing space
  - Otherwise the program will have to stop and wait!
  - Must have enough CPU to free memory
  - Allocation clock – free some memory for every allocation made; and so, ensure some free space

Generational GC

- Assumption: “Most objects live a very short time, while a small percentage of them live much longer.”
- If an object survives one collection, it will probably survive many collections
  - In a copy-collector, it will be copied over and over
- These premises lead to the Generational approach, which segregates objects by age

Generational GC Principles

- Divide the heap into parts, each with different ages (*generations*)
- When an object lives long enough, it is advanced to the next generation
- Younger generations are collected often
  - To do this, we must track pointers from the old generations to younger ones
- Usually implemented with Copy Collector

Younger Generation

ROOT SET

Older Generation

Conservative Approximation

- Generational GC is somewhat conservative
- Because it uses all intergenerational pointers as roots
- Older objects may have died, but we won't know until we collect the older generation
- So some garbage will be retained

Questions in Generational GC

- Advancement Policy – when do you get old?
- How should the heap be organized?
  - Make different spaces for each generation (copy), or use header to mark age (non-copy)?
- When should collection be scheduled?
- How to maintain intergenerational references?
  - Use write barrier? Dirty bits?
  - Many low-level options – tables, lists, pages…

Advancement Policy

- Advance young objects in every collection
  - Easy to implement; in copy-collector, saves space
  - Danger of filling older generation too fast
- Wait one collection before advancing
  - Delays advancement
  - Similar to allocating black in incremental
- Waiting longer than two cycles doesn’t seem to improve performance
- Advancement vs. Number of Generations

Pitfalls of Generational GC

- “Pig in the Snake”
  - Big data structure that lasts, and dies at once
- Small Heap Objects
  - Lots of pointers from old to young
- Large Root Set
  - Many stack and global variables must be scanned each time to check for pointers
  - Same problem as incremental…

Incremental-Generational GC – "An Unhappy Marriage"

- Generational GC uses heuristics, while Real-Time requires worst-case guarantees
  - If programmer can provide object lifetime guarantees, Generational will be effective
  - Alternatively, soft RT can tolerate some delays
- One collection is going on at any given time, which collects some generations
  - Youngest generation, or youngest two, etc.
  - When all are being collected, it's incremental GC
    - Degrades performance – more CPU or more space

Language Features related to Garbage Collection

- Sometimes, it can be useful for the programmer to be aware of the Garbage…
- **Weak Pointers –** do not prevent the object from being collected
  - Object will be retained as long as a non-weak pointer points to it, as well
  - Useful for cache-like data, or for metadata which is only relevant as long as the object lives

Language Features related to Garbage Collection – Cont.

- **Finalization** – actions to be performed automatically when the object is collected
  - Like a Destructor in C++, e.g. for closing connections or files
  - Method called asynchronously, at a time undetermined by the programmer – use carefully!
- **Multiple Heaps** – it may be useful to hold more than one heap
  - Some Garbage-Collected, others explicit
  - Or, different-policy heaps for distributed systems

Summary

- Garbage Collection is much better than explicit storage, and today it's affordable
- Basic Techniques give the building blocks
- Advanced Techniques allow state-of-the-art implementation
  - Generational GC lowers costs
  - Incremental GC allows Real-Time
- GC-related language features allow smart interaction with GC

The End…

Shai and Shiri

If you're bored

Quick Overview – Incremental Algorithms

- Incremental Copying
  - Cycle starts with a flip between the spaces
    - Objects in fromspace are unreached
    - Reachable objects will be copied “on-demand”
  - Read Barrier – any “white” object (in tospace) accessed will first be copied, only then accessed
  - In parallel, finds and copies other reachable data
  - Gray-Wave keeps one step ahead of the program
- Somewhat Conservative, and expensive
- A Non-Copying version also exists

Quick Overview – Incremental Algorithms

- Replication Copying
  - The reverse – program keeps seeing the tospace, and flips when all objects were copied
  - i.e., program sees the old versions till cycle ends
  - Write Barrier – *any update* to objects must be recopied, to keep in sync
  - Is impractical – EXCEPT! – Functional languages, where side effects are rare or nonexistent

Collecting Older Generation

- It is possible to collect older generation independently, in one of two ways:
- Recording Young-to-Old pointers
  - More costly – generally many such pointers
- Considering all young objects as root set
  - Somewhat feasible

- Usually, younger generations are collected whenever older ones are

Future research about GC

- Persistent Object Storage
- Multiprocessor – GC running in parallel
- Distributed – Data Integrity, recovery
- Fine-Grained Incremental (for multimedia, VR)
- Interoperability among systems