A Real-Time Garbage Collector Based on the Lifetimes of Objects

Henry Lieberman and Carl Hewitt
(CACM, June 1983)

Maria Jump
CS395T: Memory Management Hierarchies
September 9, 2003

Lifetime Observation

- Programs have two types of objects
  - “thinking objects” → short-lived
  - “decision objects” → long-lived
- Improve performance of Baker’s Algorithm
  - optimize for scavenging short-lived objects
  - scavenge long-lived objects less frequently
- Rental cost
  - storage management cost for an object is proportional to the time during which the object is used

Modify Baker’s Algorithm

- address space broken up into generations
  - small (relative to address space) set of pages
- Baker’s algorithm used per region
  - flip → condemning a region
  - fromspace → obsolete areas
  - tospace → non-obsolete areas
  - evacuate objects in the same way
Regions are tagged with generation and version number

Allocation occurs in **creation region**

... until the **creation region** is full
... then new creation region is created

... and so on

... and so on
... and so on

GC is initiated by condemning a region (making it obsolete) and creating an evacuation region

this requires that younger generations also be condemned
accessible objects in condemned region(s) are incrementally evacuated using Baker’s Algorithm

... and the memory is recycled

The Good – Varying GC Rates

- Vary the rate of GC for each generation
  - younger generations contain high percentages of garbage and are collected frequently
  - older generations contain relatively permanent data and are collected seldomly
- Focus scavange time where the highest proportion of inaccessible objects
- Minimizes the amount of scavenging needed per inaccessible object
The Bad – Fragmentation

- Fragmentation results from partially-filled regions
  - choose a region size to minimize fragmentation
  - coalesce older regions reducing the amount of wasted space

The Ugly – Scavenge Time

- Whole heap scavenge required to ensure no pointers point into condemned region(s)
- Scavenging is a lot of work
- Scavenging is good because it
  - reuses the address space
  - compacts the address space
- Want to reduce the scavenging time by using forward pointers

Forward Pointers
Forward Pointers

- CONS composes existing components into objects
  - creates a backward pointer

Forward Pointers

- CONS composes existing components into objects
  - creates a backward pointer
- RPLACA can assign a new pointer as a component to an older object
  - creates a forward pointer

Entry Tables

- Problem occurs after heap is manipulated further
Problem occurs after heap is manipulated further
Add entry table to record forward pointers

Problem occurs after heap is manipulated further
Add entry table to record forward pointers
Add a level of indirection for some pointers
GC uses entry table entries as roots to the region
Contributions

- Different objects have different lifetimes
- Performance benefit from varying the collection of rates of objects with different lifetimes
- Introduced concept of different generations within a copying collector
- Introduced use of entry table to avoid scanning entire heap

Sidebar 1: Weak Pointers

Pointers in the program which do not protect an object from being collected
- not followed during GC
- small in number
- forward weak pointers use the entry table

Sidebar 2: Value Cells & Stacks

Represent roots of scavenging
- What is the advantage of considering them the “oldest” generation?
- What is the advantage of considering them the “youngest” generation?
  - no entry tables pointers needed
Sidebar 3: Flavors of Allocation

Providing different allocators for different regions

- Can sophisticated users direct the flavor of allocation?
- Can compiler analysis accurately change the flavor of allocation?
- Can runtime information accurately change the flavor of allocation?

Aspects of Program Behavior

- Rate of object creation
- Average lifetimes of objects
- Proportion of forward vs. backward pointers
- Average “length” of pointers

The End