University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Garbage Collection (2)

Advanced Operating Systems

Lecture 8

Colin Perkins | https://csperkins.org/ | Copyright © 2017 | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Lecture Outline

- Garbage collection
  - …
  - Generational algorithms
  - Incremental algorithms
  - Real-time garbage collection
- Practical factors

Object Lifetimes

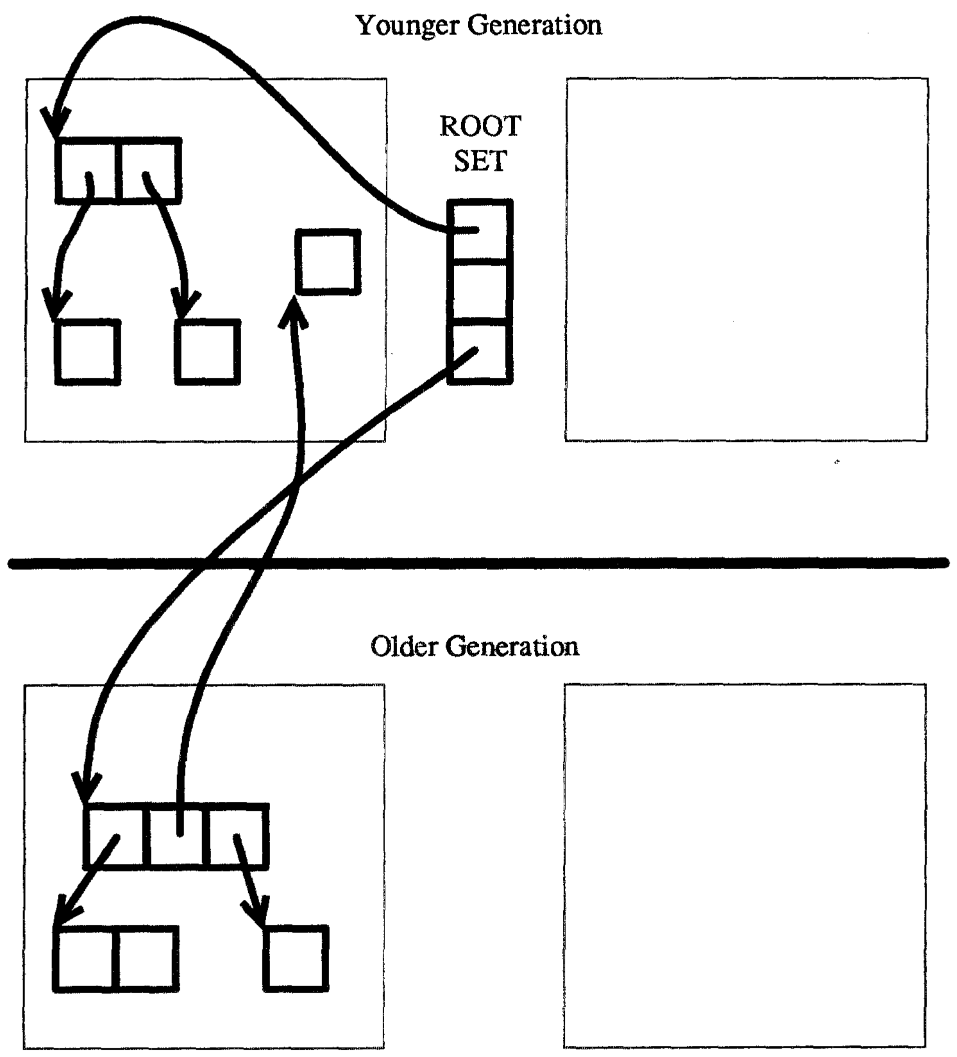
- Studies have shown that most objects live a very short time, while a small percentage of them live much longer
  - This seems to be generally true, no matter what programming language is considered, across numerous studies
  - Although, obviously, different programs and different languages produce varying amount of garbage

- Implication: when the garbage collector runs, live objects will be in a minority
  - Statistically, the longer an object has lived, the longer it is likely to live
  - Can we design a garbage collector to take advantage?

Colin Perkins | https://csperkins.org/ | Copyright © 2017

A Copying Generational Collector (1)

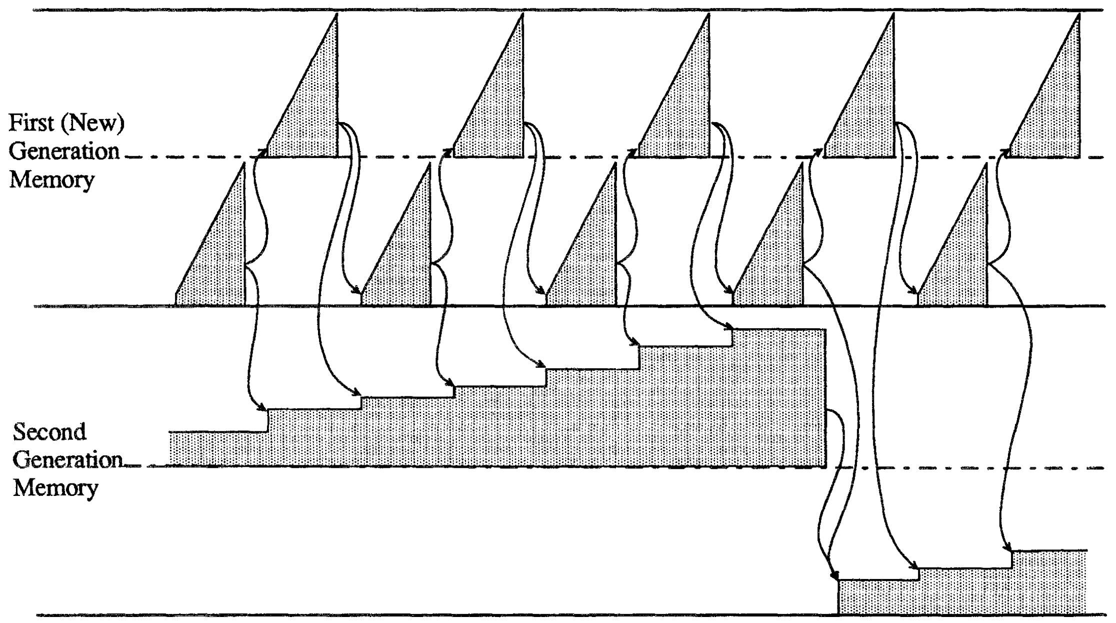
- In a generational garbage collector, the heap is split into regions for long-lived and young objects
  - Regions holding young objects are garbage collected more frequently
  - Objects are moved to the region for long-lived objects if they're still alive after several collections
  - More sophisticated approaches may have multiple generations, although the gains diminish rapidly with increasing numbers of generations
- Example: stop-and-copy using semispaces with two generations
  - All allocations occurs in the younger generation's region of the heap
  - When that region is full, collection occurs as normal
  - …



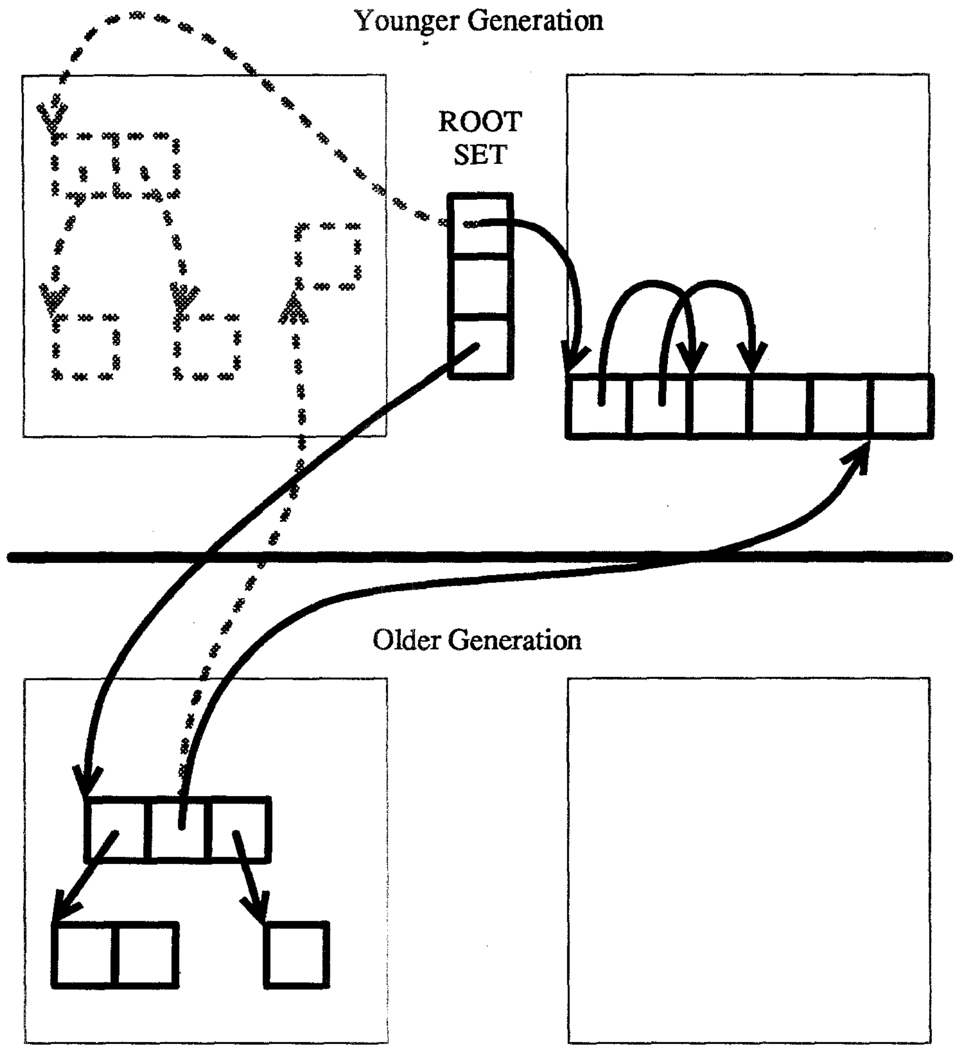
Colin Perkins | https://csperkins.org/ | Copyright © 2017

A Copying Generational Collector (2)

- …
- Objects are tagged with the number of collections of the younger generation they have survived; if they're alive after some threshold, they're copied to the older generation's space during collection
- Eventually, the older generation's space is full, and is collected as normal



- Note: not to scale: older generations are generally much larger than the younger, as they're collected much less often



Detecting Intergenerational References

- In generational collectors, younger generation must collected independent of the long-lived generation
  - But – there may be object references between the generations
  - Young objects referencing long-lived objects common but straight-forward since most young objects die before the long-lived objects are collected
    - Treat the younger generation objects as part of the root set for the older generation, if collection of the older generation is needed
  - Direct pointers from old-to-young generation are problematic, since they require a scan of the old generation to detect
  - May be appropriate to use an indirection table ("pointers-to-pointers") for old-to-young generation references
    - The indirection table forms part of the root set of the younger generation
    - Movement on objects in the younger generation requires an update to the indirection table, but not to long-lived objects
    - Note: this is conservative: the death of a long-lived object isn't observed until that generation is collected, but that may be several collections of the younger generation, in which time the younger object appears to be referenced

cc BY ND

Generational Garbage Collection

- Variations on this concept are widely used
  - E.g., the ~~Sun~~ Oracle HotSpot JVM uses a generational garbage collector

- Generational collectors achieve good efficiency:
  - Cost of collection is generally proportional to number of live objects
  - Most objects don't live long enough to be collected; those that do are moved to a more rarely collected generation
  - But – eventually the longer-lived generation must be collected; this can be very slow

Colin Perkins | https://csperkins.org/ | Copyright © 2017

Incremental Garbage Collection

- Preceding discussion has assumed the collector “stops-the-world” when it runs
  - Clearly problematic for interactive or real-time applications
- Desire a collector that can operate incrementally
  - Interleave small amounts of garbage collection with small runs of program execution
  - Implication: the garbage collector can’t scan the entire heap when it runs; must scan a fragment of the heap each time
  - Problem: the program (the “mutator”) can change the heap between runs of the garbage collector
  - Need to track changes made to the heap between garbage collector runs; be conservative and don’t collect objects that might be referenced – can always collect on the next complete scan

Colin Perkins | https://csperkins.org/ | Copyright © 2017

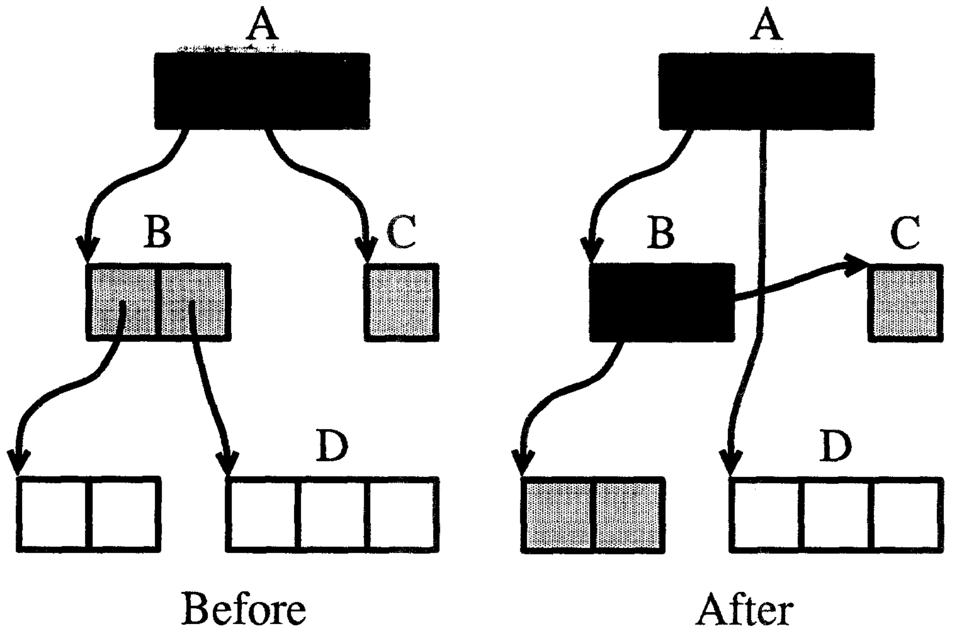
Tricolour Marking

- For each complete collection cycle, each object is labelled with a colour:
  - White – not yet checked
  - Grey – live, but some direct children not yet checked
  - Black – live
- Basic incremental collector operation:
  - Garbage collection proceeds with a wavefront of grey objects, where the collector is checking them, or objects they reference, for liveness
  - Black objects behind are behind the wavefront, and are known to be live
  - Objects ahead of the wavefront, not yet reached by the collection, are white; anything still white once all objects have been traced is garbage
  - No direct pointers from black objects to white – any program operation that will create such a pointer requires coordination with the collector

Colin Perkins | https://csperkins.org/ | Copyright © 2017

Tricolour Marking: Need for Coordination

- Garbage collector runs
  - Object A scanned, known to be live → black
  - Objects B and C are reachable via A, and are live, but some of their children have not been scanned → grey
  - Object D not checked → white
- Program runs, and swaps the pointers from A→C and B→D such that A→D and B→C
- This creates a pointer from black to white
  - Program must now coordinate with the collector, else collection will continue, marking object B black and its children grey, but D will not be reached since children of A have already been scanned



Coordination Strategies

- Read barrier: trap attempts by the program to read pointers to white objects, colour those objects grey, and let the program continue
  - Makes it impossible for the program to get a pointer to a white object, so it cannot make a black object point to a white
- Write barrier: trap attempts to change pointers from black objects to point to white objects
  - Either then re-colour the black object as grey, or re-colour the white object being referenced as grey
  - The object coloured grey is moved onto the list of objects whose children must be checked

cc BY ND

Incremental Collection

- Many variants on read- and write-barrier tricolour algorithms
  - Performance trade-off differs depending on hardware characteristics, and on the way pointers are represented
  - Write barrier generally cheaper to implement than read barrier, as writes are less common in most code
- There is a balance between collector operation and program operation
  - If the program tries to create too many new references from black to white objects, requiring coordination with the collector, the collection may never complete
  - Resolve by forcing a complete stop-the-world collection if free memory is exhausted, or after a certain amount of time

BY ND

Real-time Garbage Collection

- Real-time collectors build incremental collectors
  - Two basic approaches:
    - Work based: every request to allocate an object or assign an object reference does some garbage collection; amortise collection cost with allocation cost
    - Time based: schedule an incremental collector as a periodic task
  - Obtain timing guarantees by limiting amount of garbage that can be collected in a given interval to less than that which can be collected
  - The amount of garbage that can be collected can be measured: how fast can the collector scan memory (and copy objects, if a copying collector)
    - Cannot collect garbage faster than the collector can scan memory to determine if objects are free to be collected
    - This must be a worse-case collection rate, if the collector has varying runtime
  - The programmer must bound the amount of garbage generated to within the capacity of the collector

Bacon *et al*. A real-time garbage collector with low overhead and consistent utilization. Proc. ACM symposium on Principles of programming languages, 2003, New York. DOI 10.1145/604131.604155

cc BY ND

Practical Factors

- Two significant limitations:
    - Interaction with virtual memory
    - Garbage collection for C-like languages

- In general, garbage collected programs will use significantly more memory than (correct) programs with manual memory management
    - E.g., many of the copying collectors must maintain two regions, and so a naïve implementation doubles memory usage

Colin Perkins | https://csperkins.org/ | Copyright © 2017

Interaction with Virtual Memory

- Virtual memory subsystems page out unused data in an LRU manner
- Garbage collector scans objects, paging data back into memory
- Leads to thrashing if the working set of the garbage collector larger than memory
  - Open research issue: combining virtual memory with garbage collector

cc BY ND

Garbage Collection for C-like Languages

- Collectors rely on being able to identify and follow pointers, to determine what is a live object
- C is weakly typed: can cast any integer to a pointer, and can do arithmetic on pointers
  - Implementation-defined behaviour, since pointers and integers are not guaranteed to be the same size
- Greatly complicates garbage collection:
  - Need to be conservative: any memory that might be a pointer must be treated as one
  - The Boehm-Demers-Weiser garbage collector can be used for C and C++ (http://www.hpl.hp.com/personal/Hans_Boehm/gc/) – this works for strictly conforming ANSI C code, but beware that much code is not conforming

Further Reading

- Bacon *et al.*, "A real-time garbage collector with low overhead and consistent utilization", Proc. ACM Principles of Programming Languages, 2003, New York. DOI:10.1145/604131.604155

- To consider:
  - Problems and limitations of prior work
  - Operation of the real-time garbage collector
  - Real-time scheduling
  - Practical factors and implementation considerations

A Real-time Garbage Collector with Low Overhead and Consistent Utilization

David F. Bacon
dfb@watson.ibm.com

Perry Cheng
perryche@us.ibm.com

V.T. Rajan
vtrajan@us.ibm.com

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT

Now that the use of garbage collection in languages like Java is becoming widely accepted due to the safety and software engineering benefits it provides, there is significant interest in applying garbage collection to hard real-time systems. Past approaches have generally suffered from one of two major flaws: either they were not provably real-time, or they imposed large space overheads to meet the real-time bounds. We present a mostly non-moving, dynamically defragmenting collector that overcomes both of these limitations: by avoiding copying in most cases, space requirements are kept low; and by fully incrementalizing the collector we are able to meet real-time bounds. We implemented our algorithm in the Jikes RVM and show that at real-time resolution we are able to obtain mutator utilization rates of 45% with only 1.6–2.5 times the actual space required by the application, a factor of 4 improvement in utilization over the best previously published results. Defragmentation causes no more than 4% of the traced data to be copied.

General Terms

Algorithms, Languages, Measurement, Performance

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Java; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

Keywords

Read barrier, defragmentation, real-time scheduling, utilization

1. INTRODUCTION

Garbage collected languages like Java are making significant inroads into domains with hard real-time concerns, such as automotive command-and-control systems. However, the engineering and product life-cycle advantages consequent from the simplicity of programming with garbage collection remain unavailable for use in the core functionality of such systems, where hard real-time constraints must be met. As a result, real-time programming requires the use of multiple languages, or at least (in the case of the Real-Time Specification for Java [9]) two programming models within the same language. Therefore, there is a pressing practical need for a system that can provide real-time guarantees for Java without imposing major penalties in space or time.

We present a design for a real-time garbage collector for Java, an analysis of its real-time properties, and implementation results that show that we are able to run applications with high mutator utilization and low variance in pause times.

The target is uniprocessor embedded systems. The collector is therefore concurrent, but not parallel. This choice both complicates and simplifies the design: the design is complicated by the fact that the collector must be interleaved with the mutators, instead of being able to run on a separate processor; the design is simplified since the programming model is sequentially consistent.

Previous incremental collectors either attempt to avoid overhead and complexity by using a non-copying approach (and are therefore subject to potentially unbounded fragmentation), or attempt to prevent fragmentation by performing concurrent copying (and therefore require a minimum of a factor of two overhead in space, as well as requiring barriers on reads and/or writes, which are costly and tend to make response time unpredictable).

Our collector is unique in that it occupies an under-explored portion of the design space for real-time incremental collectors: it is a *mostly non-copying* hybrid. As long as space is available, it acts like a non-copying collector, with the consequent advantages. When space becomes scarce, it performs defragmentation with limited copying of objects. We show experimentally that such a design is able to achieve low space and time overhead, and high and consistent mutator CPU utilization.

In order to achieve high performance with a copying collector, we have developed optimization techniques for the Brooks-style read barrier [10] using an "eager invariant" that keeps read barrier overhead to 4%, an order of magnitude faster than previous software read barriers.

Our collector can use either time- or work-based scheduling. Most previous work on real-time garbage collection, starting with Baker's algorithm [5], has used work-based scheduling. We show both analytically and experimentally that time-based scheduling is superior, particularly at the short intervals that are typically of interest in real-time systems. Work-based algorithms may achieve short individual pause times, but are unable to achieve consistent utilization.

The paper is organized as follows: Section 2 describes previ-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.
Copyright © 2003 ACM 1-58113-628-5/03/0001 $5.00.

Colin Perkins | https://csperkins.org/ | Copyright © 2017