Simple Garbage Collection and Fast Allocation

Andrew W. Appel

Presented by

Karthik Iyer

Agenda

- Background
- Motivation
- Appel’s Technique
- Terminology
- Fast Allocation
- Arranging Generations
- Invariant
- GC Working
- Heuristic
- Handling Assignments
- Handling Registers
- Conclsion
- Questions

Background

- Modern Computers – Automatic GC
- Mark and Sweep
  - Parses and marks the object graph – DFS
  - Unmarked objects in free list
  - O(size of mem)
- Copy Collectors
  - Copies live objects from src to dst regions
  - Src and dst flip
  - O(live object mem)

Background contd…

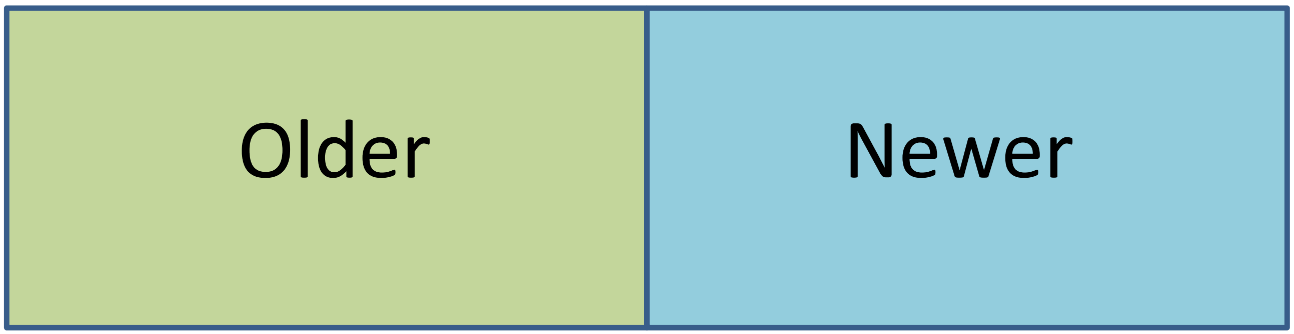
- Key observations
  - Newer objects point to old ones
  - Young object more likely to become garbage
- Generational GC
  - Memory divided into regions
  - Objects of similar age – same region
  - GC of region x => GC of regions y < x
  - Not suitable for all languages
- Handling 'Reverse Pointers'
  - Entry Table – Lieberman & Hewitt
  - Special VM Hardware, marking pages – Moon
  - List of pointers - Ungar
- Cheney's Algorithm
  - If a root R points in dst, object already copied
  - Else, copy the object, store a forwarding pointer, update NEXT and SCAN

Motivation

- In a Generational GC Setup, lower bound for reclamation is very low, close to zero – Why?
  - All the objects in the collected region are dead
  - Allocation may take more time then collection
  - Technique for fast allocation
- When to ask memory from OS?
  - Need a way to organize generations
  - Also a Heuristic to aid decisions

Appel's Technique

- Divide heap into two regions – Copy from Newer to Older



- Why only two regions?
  - Two generations maximize space for the allocator
  - Large space reduces GC cycles, avoids unnecessary elevation of objects, reduces copying
  - Other (will visit later)

Appel’s Technique contd…

- Uses Cheney’s algorithm
- A technique for fast allocation
- A way to layout heap
- Heuristic to aid memory management
  - based on mutator behavior
- A way to keep track of ‘reverse pointers’
- Unix implementation

Terminology

- Assignment – A modification to the records in the older region such that he record is made to point to a newer region
- Minor Collection – GC of the Newer Region. Appends live objects into Older Region.
- Major Collection – GC of the Older Region. Retains live objects in the Older Region
- Free Space Pointer (FSP) – Pointer to first free byte in the Newer Region
- Free Space Limit – End of the Newer Region
- Inaccessible Page – A dedicated page access to which triggers GC as Newer Region is full

Fast Allocation

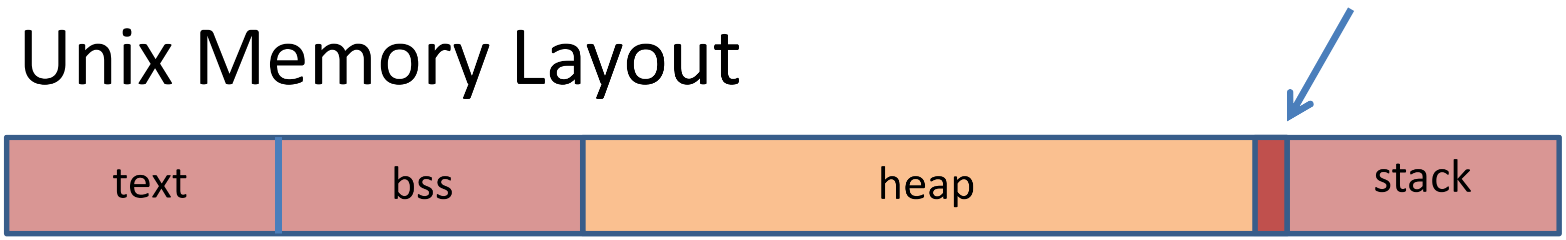
- Unallocated region is continous
- CONS(A,B)
  - Test FSP below Limit, trigger GC if at limit
  - FSP = FSP -2
  - Store A, B and return FSP
- Overhead of CONS
  - Test executed by VM handler. Page Fault if FSP hits inaccessible page
  - On a VAX, auto decrement FSP while store
  - Hence overhead Zero

Variable Sized Records

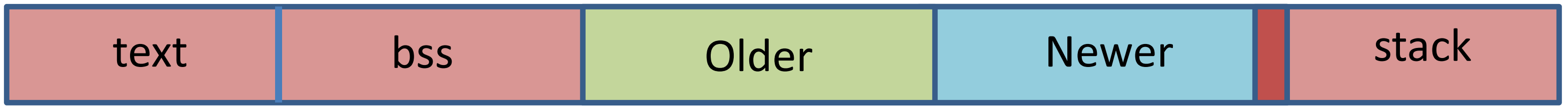
- Many techniques to handle
  - Associate tag with object
  - Allocate objects in different regions based on size
  - Type System map in a statically typed language
  - We use tags
- Issues
  - Allocation is difficult, may span beyond region
  - Two word records not an issue as regions boundaries are even
- Solution
  - Allocate in reverse order
  - If last word fits, rest is guaranteed to fit
  - Page Fault if last word hits inaccessible page

Arranging Generations

- Arrange heap such that inaccessible page at one end of heap
- Unix Memory Layout



- Inaccessible Page – Unix Program Break
- Newer Region – Starts mid-heap and grows towards inaccessible page
- Older Region – Starts beginning of heap and grows towards mid-heap point

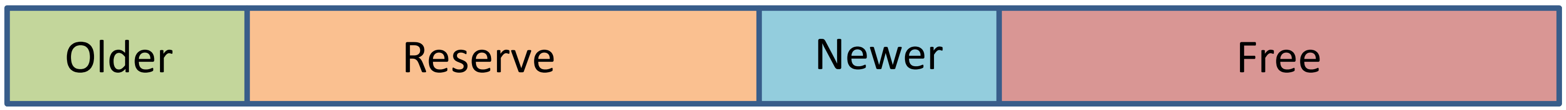


Invariant

- A two region copying collector runs out of space if there is not enough free space in dst region to fit src region live objects
- Hence $M > 2A$
  - where M – heap size, A – mutator requirement
- If A approaches M/2, GC performance degrades – Why?
  - Too many collections
- For good performance, $M >> 2A$

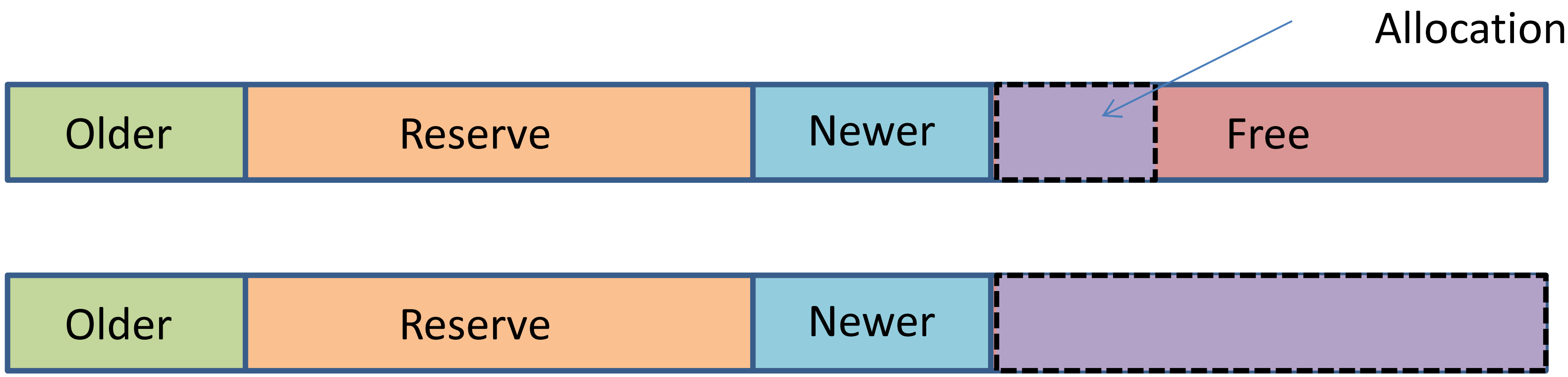
GC Working

- Heap Organization

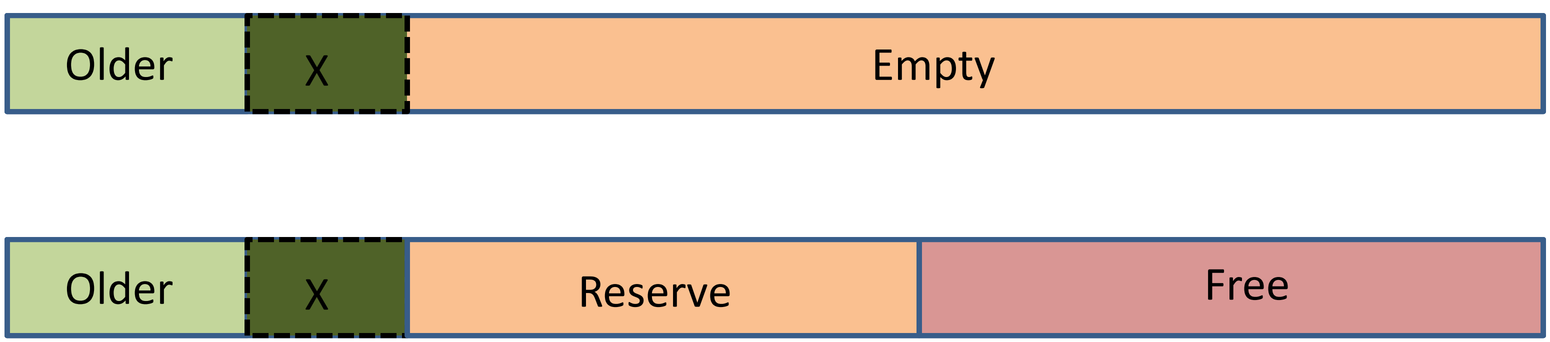


- Reserve – Free space in Older Region
- Free – Free space in the Newer Region
- Working
  - Allocation happens in Free area until inaccessible page is hit (Minor Collection - MiC Triggered)
  - MiC copies live data (x) at the end of Older Region
  - Remaining free space beyond x, is divided into equal regions to accommodate Reserve and Free regions

GC Working - Illustration



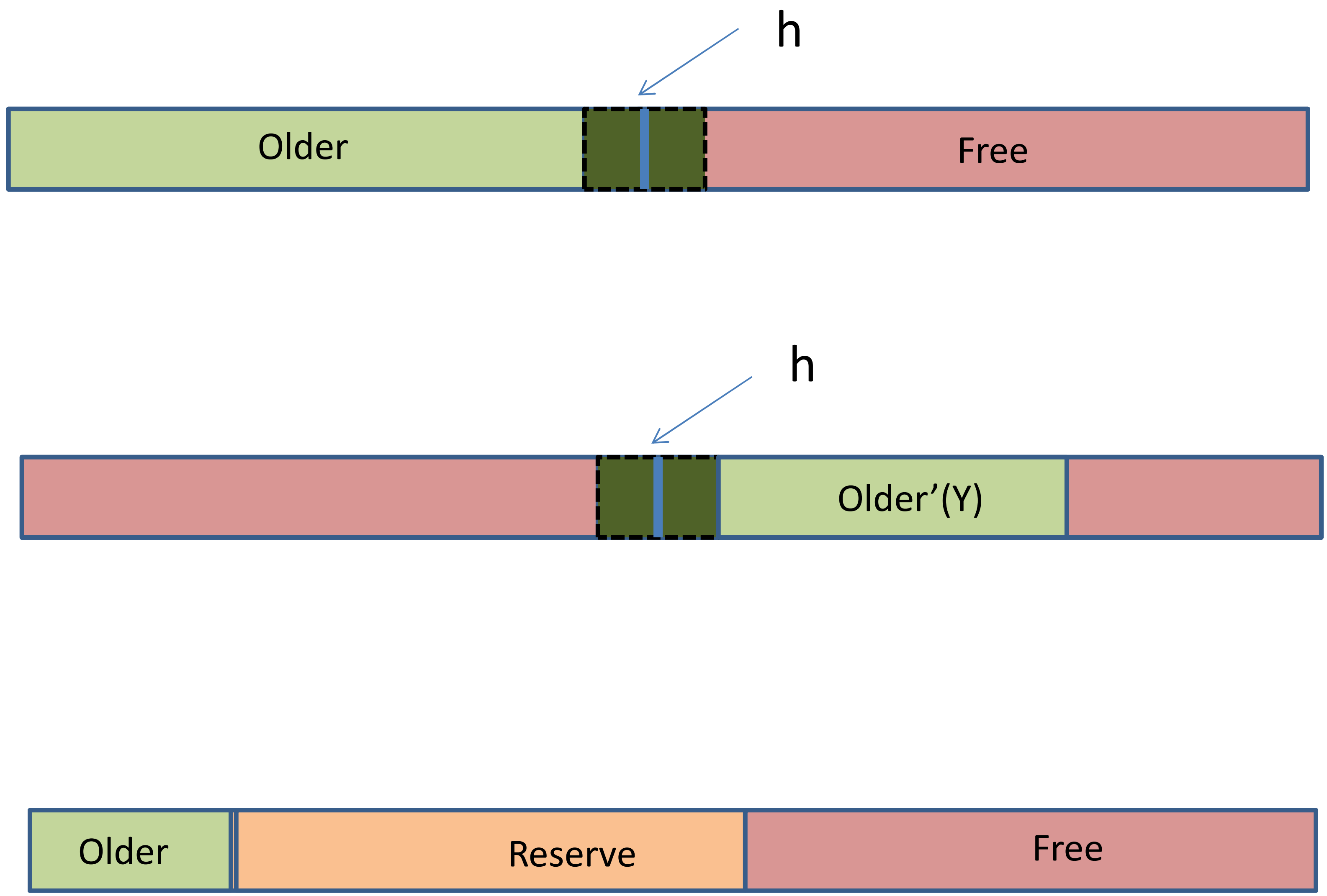
Minor Collection Triggered



GC Working contd….

- At some point, Older region fills out – appending ‘X’ of a MiC crosses the Older Region Boundary (h)
- Major Collection (MaC) Triggered
- How do we ensure enough space for live objects in Older Region (Y)?
  – Copy Y after X
  – Since we crossed h after appending X to Y, Y < M/2
  – Free Space = M/2, hence Y fits
- Bring back Y+X to beginning of heap, call it Older Region

GC Working – Illustration contd…



When to ask for Memory? Heuristic !

- Let Υ be M/A
- If $\Upsilon < 2$, mutator runs out of space
- If $\Upsilon = 2$, GC performance degrades
- $\Upsilon >> 2$
  - Depending on compiler and language GC performance varies based on Υ
  - But higher Υ => better GC performance
- Let Υ' be required Υ for required performance

Heuristic contd…

- Allocated some space initially – may be based on predicted value of A
- If A grows – ask for more memory from OS
- How to know A is growing? When to trigger?
  - After MaC, if $\Upsilon' < \Upsilon$
  - No space when appending Y to X. $\Upsilon < 2$
  - After an MiC, Free region not enough for a huge object. Most probably $A > M/\Upsilon'$
- Use UNIX brk() for more memory
- Calculating A, hence Υ in a multi-generation setup is difficult

Keeping track of Assignments

- Root Set – Globals, Registers, Stack and Reverse Pointers
- Stack in heap saves tracking stack frames – efficient
- How to keep track of assignments?
  - A linked list or a list of vectors of all Reverse Pointers
  - Compiler adds code to insert Reverse Pointers to the list during assignment
  - Compiler runs checks to see if an assignment is a Reverse Pointer

Assignments contd…

- Multi-generational setup – After each MiC, need to maintain pruned lists. Difficult
  - Remember our Two region setup? List will empty after each Mic. Advantage.
- Root list overhead
  - 8 instructions
  - 4 for list update
  - 4 to examine a record
  - List Traversal overhead to check duplicate?

Handling Registers

- Registers are part of Root set
- During Traps and GC, Root set modified
- But Traps require registers for execution
- Root Set registers pushed onto stack and modified
- Can run a check to learn about what registers pushed and the order

Conclusion

- Fast allocation
- Simple Algorithm – Two regions
- No entry tables – Reverse pointers part of heap. Less overhead to manage these pointers
- Easy to implement in UNIX

Questions

- What does fast mean? How many instructions other collectors take to allocate? Compare.
- Isn't fast too tied to a machine?
- Assessing A may be skewed due to list of reverse pointers in heap?
- Why size of Free Space reduced after each MiC?