CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107S in the
Winter 2012/2013 term at the University of Toronto

Dynamic Memory Management

- Management of dynamically created runtime storage is a central issue in the implementation of many programming languages.
- Most implementations allocate storage in a *heap* with some mechanism for reclaiming memory that is no longer in use.
- Garbage collection[a] is a tool for automatically managing dynamically allocated memory.
  - Identify dynamically allocated memory that is no longer in use.
  - Recycle that memory for later reuse
- Some programming language require Garbage Collection (Java, Lisp), many others allow it.
- The presence of Garbage Collection adds runtime overhead and may introduce unpredictable *stalls* to program execution

[a] Richard Jones, Antony Hoskin, Elliot Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, Chapman & Hall, 2011

Dynamic Storage Design Issues

- Roots and pointer finding.
  Is it possible to find all pointers (roots) to allocated objects?
  This is really hard to get correct.
- Mobility of data. It is only possible to move a data object if it is possible to (eventually) update *all pointers* to the object to point at the new location.
- Immediacy of storage reclamation. Is recyclable memory made available immediately or only periodically.
- Program tolerance to interruption and/or arbitrary delay.
  Can mutation and collection proceed concurrently?
- Processing Cost – Time and Space Overhead
- Fragmentation of heap memory.
  Performance Degradation as heap utilization increases.
- Recovering cyclic data structures.

Basic Principle

- The key concept for garbage collection is reachability. A program can only access some object if it has a memory address (i.e. pointer) for the object. An object that has been allocated but is no longer reachable, is *garbage* and may be recycled.
- *Roots* are base locations that may hold the address of dynamically allocated objects. Possible roots include: processor registers, all kinds of variables and temporaries in activation records.
- Dynamically allocated memory can only be accessed through some chain of pointers starting with some root. An object is *reachable* if and only if there is some path from a root to the object.
- Two separate issues:
  – identifying reachable objects
  – managing and recycling memory efficiently.

Classic Dynamic Memory Design Approaches

- *Reference Counting* - Every allocated object contains a *reference count*. This count is incremented when a new pointer to the object is created and decremented when a pointer to the object is destroyed. A reference count of zero implies the object has no pointers pointing at it and so it can be reclaimed. Incremental algorithm.
- *Mark and Sweep Collection* - A garbage collector algorithm starts at all roots recursively marking all nodes that can be reached. When this algorithm finishes, any nodes that are not marked can be reclaimed.

  Conservative marking , takes more space, but use a simpler algorithm to identify pointers. Strategy:
  - Sweep all available memory, test each word for the *isPointer* property
  - Unless the collector can (heuristically) decide that a word does not contain a pointer, assume that it is a pointer.
  - If the collector knows the range of valid memory addresses, it can efficiently exclude, small integers, real numbers and strings

Compacting and Non-compacting Collectors

- Once recyclable memory has been identified, there are two main options for managing this memory.
- Compacting collectors move all live objects into a block of contiguous memory. This leaves a large block of memory available for reuse.
- Non-compacting collectors leave the live objects in place, and (somehow) link all the recycled memory for later use.
- Compacting advantages
  - Little effort required to reclaim objects.
  - New allocations are easy with a large block of contiguous free memory.
  - Moving object together may improve cache performance.
- Compacting Disadvantages
  - All pointer access must be managed through the collector.
  - Moving large blocks of memory may impact memory performance.
  - May require twice as much memory for compacting.

Reference Counting

- Used in many languages and applications:
  Smalltalk, InterLisp, Java, Adobe PhotoShop
- Storage management runs concurrently intermixed with program execution.
- Invariant: the reference count of each node is equal to the number of pointers pointing to the node.
- Can do optimizations on pointers local to a routine to reduce the amount of reference counting.

Reference Counting - Advantages

- Memory management overhead is distributed throughout the computation. May yield smoother response time.
- Locality of reference no worse than application.
- Data from real languages suggests that few nodes are shared and many nodes have a short lifetime. Reference counting lets these nodes be reused as soon as they are free.
  This may lead to better cache/virtual memory behavior.
  This may allow *update in place* optimizations.
- Performance relatively independent of heap occupancy fraction.

Reference Counting - Disadvantages

- High processing cost. Every pointer assignment requires reference count updating.
- Correctness is difficult.
  Missing even one reference counter update could lead to errors.
- Difficult to deal with embedded pointer assignments,
  e.g. bulk assignment of a structure containing pointers.
- Reference count roughly doubles the size of every pointer.
- Reference counting schemes can not reclaim cyclical data structures, e.g. a doubly linked list.
- Need some strategy to deal with reference counter overflow.

Naive Mark Sweep Algorithm[a]

- Basic algorithm:
  - Suspend program processing.
  - Start at all roots. The algorithm must be able to find **all** roots, e.g. even roots in registers.
  - Trace nodes reachable from the roots and mark them.
  - If trace reaches a node that is already marked, stop tracing on that path.
  - After marking is complete, make another sweep through the heap
    unmarked nodes can be reclaimed.
    reset mark on marked nodes for next sweep.
  - Restart program execution.
- Mark and Sweep is usually triggered asynchronously when the program makes a request for memory that cannot be satisfied.

[a] Used in original Lisp implementation.

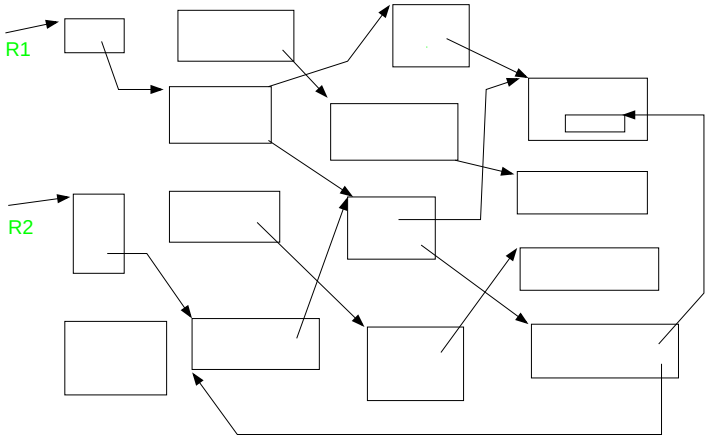
Mark and Sweep Advantages

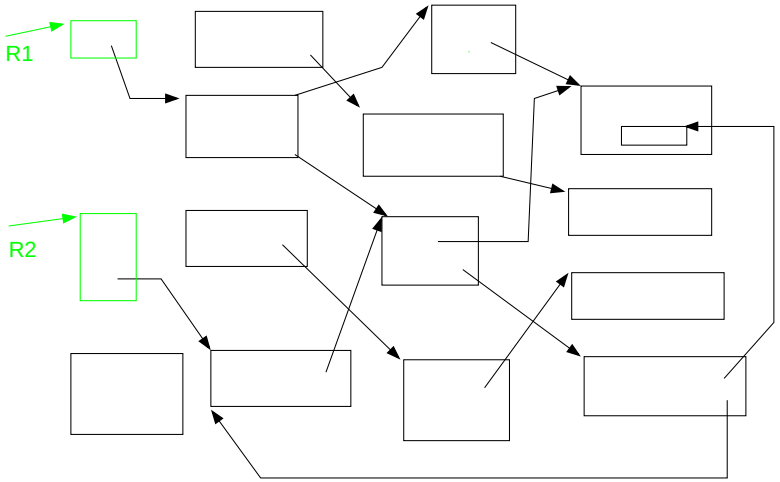
- Cycles handled easily.
- No overhead on pointer manipulation.
- No space overhead on pointers.
- Handles bulk assignment correctly.

Mark and Sweep - Disadvantages

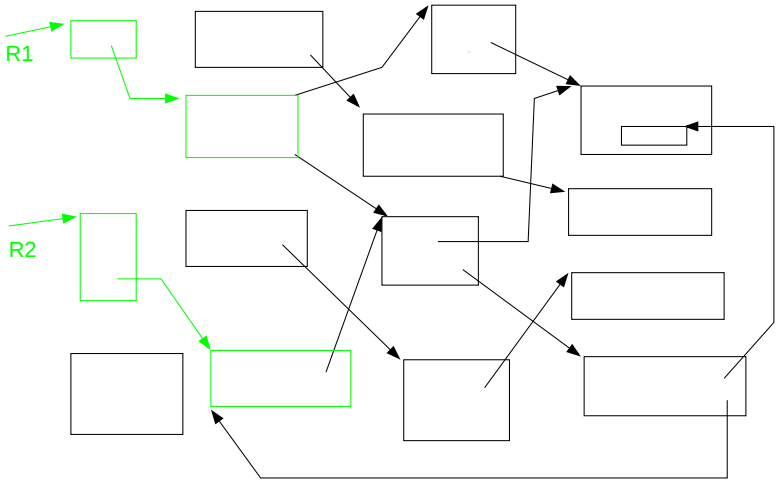
- Stop - Start algorithm.
  Unsuitable for interactive applications.
- Guaranteeing that all roots can be found is *hard*
- Need to disable collector in critical sections.
- Cost of mark/sweep passes is high especially in modern memory hierarchies.
- Time is proportional to size of heap, not number of active cells.
  Algorithm will tend to *thrash* as heap occupancy increases.
- Without optimizations leads to highly fragmented memory.

Naive Mark and Sweep Example

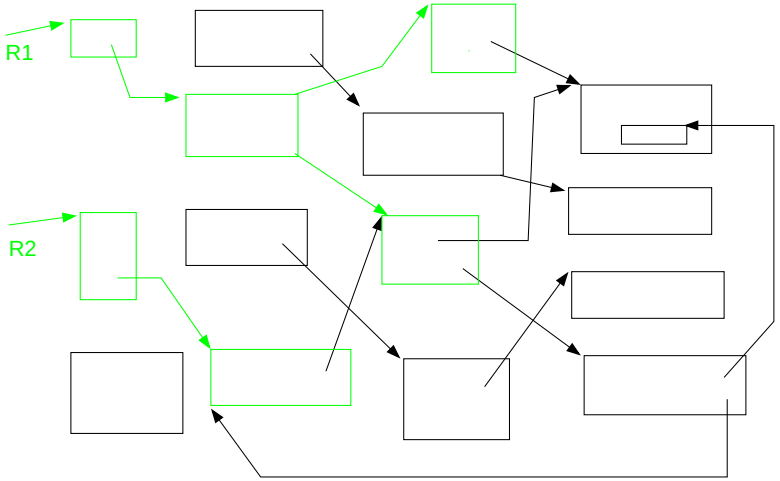




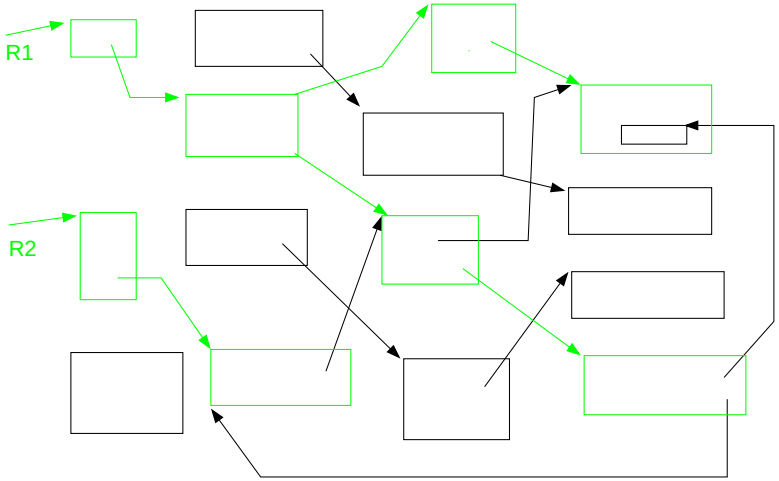
323



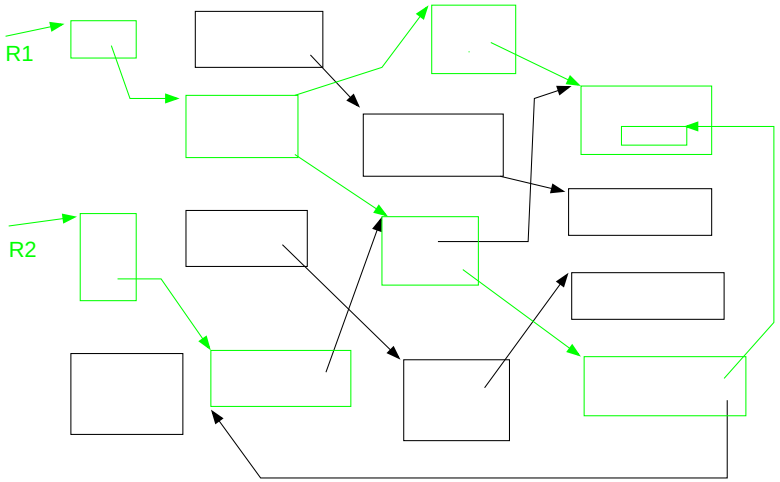
324

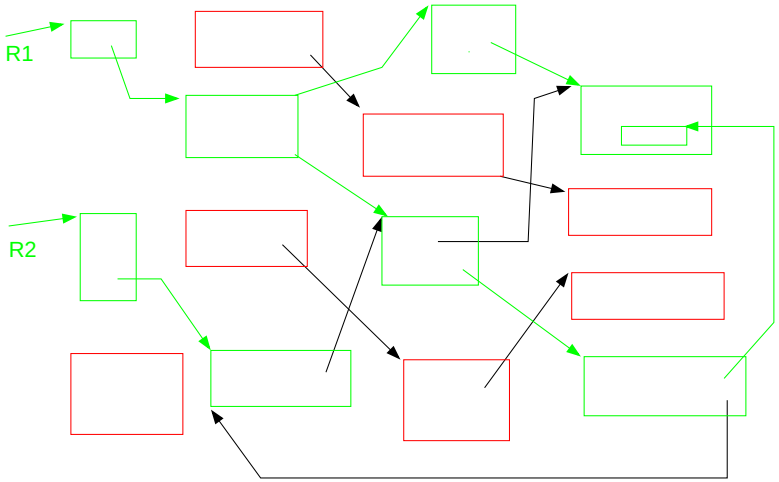


325



326





Tri-color Marking

- A modern version of Mark/Sweep. This is an *on the fly* algorithm.
  Allocation and mutation can continue during collection.
- Start with three sets:
  - White set contains objects that can have their memory recycled.
  - Black set contains objects that have no pointers to objects in the white set but are not available for recycling.
  - Gray set contains all objects reachable from root references.
    Objects referenced from the grey set haven't been scanned yet.
    Grey set objects will eventually move to the Black set.
- Initially
  - The Black set is empty
  - The Grey set is initialized to all objects that can be reached *directly* from roots
  - The White set contains all objects not in the grey set.

- Algorithm
  - Objects only move from white to grey to black.
  - Pick an object (*victim*) from the Grey Set.
    Move all objects in the White set that *victim references directly* to the Grey set.
    Move object to the Black Set.
  - Iterate until the Grey set is empty.
    Then all objects in the White set can be recycled.
- **Invariant** No object in the Black Set points directly to an Object in the White set.
  Any allocation or mutation during collection must maintain this invariant.

Compacting Collectors

- Widely used due to superior performance over reference counting and mark/sweep.
- Uses two large regions of memory *fromSpace* and *toSpace*.
  One region is active, containing data objects and new allocations
- Compacting collector algorithm
  - Suspend program execution.
  - Flip identity of two semi-spaces. *toSpace* $\leftrightarrow$ *fromSpace*
  - Trace from roots all live data structures in fromSpace.
  - Copy live nodes to toSpace when the node is first visited.
  - Update all pointers on the fly to point to new location of node.
    Assume fromSpace and toSpace pointers can be distinguished.
    Requires mapping table of old/new pointer values.
  - fromSpace becomes inactive until the next collection
  - Restart program execution.

- Compacting Collection - Advantages
  - Good time performance. Efficient allocation.
  - Asymptotic complexity less than marks/sweep collectors.
  - Fragmentation is eliminated since live data is copied compactly into the toSpace.
  - Can handle bulk pointer assignment.
- Compacting Collection - Disadvantages
  - Must be able to identify and update *all* pointers to data objects being moved.
  - May degrade paging in virtual memory systems.
    Collector may get more page faults than other techniques.
  - Requires potentially large mapping table during copying to update pointers.
  - Stop - start algorithm. Not suitable for interactive or real-time systems.
  - Requires double the address space of other techniques.