Garbage Collection

Advanced Systems Programming (H/M)

Lecture 6

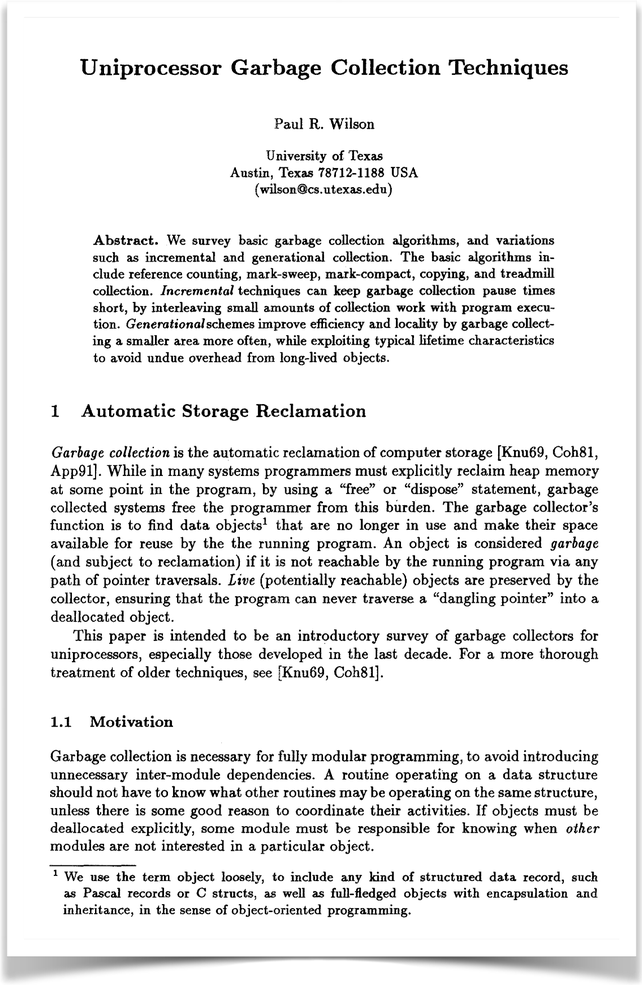
Rationale

- Region-based memory management (→ lecture 5) is novel, trades program complexity for predictable resource management
- Garbage collection widely implemented, but less predictable

- Need to understand garbage collector operation to understand the performance-complexity trade-off

Lecture Outline

- Garbage collection algorithms
  - Mark-sweep
  - Mark-compact
  - Copying collectors
  - Generational collectors
  - Incremental collectors
- Real-time garbage collection
- Practical factors



Uniprocessor Garbage Collection Techniques

Paul R. Wilson

University of Texas
Austin, Texas 78712-1188 USA
(wilson@cs.utexas.edu)

Abstract. We survey basic garbage collection algorithms, and variations such as incremental and generational collection. The basic algorithms include reference counting, mark-sweep, mark-compact, copying, and treadmill collection. *Incremental* techniques can keep garbage collection pause times short, by interleaving small amounts of collection work with program execution. *Generational* schemes improve efficiency and locality by garbage collecting a smaller area more often, while exploiting typical lifetime characteristics to avoid undue overhead from long-lived objects.

1 Automatic Storage Reclamation

Garbage collection is the automatic reclamation of computer storage [Knu69, Coh81, App91]. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a "free" or "dispose" statement, garbage collected systems free the programmer from this burden. The garbage collector's function is to find data objects[1] that are no longer in use and make their space available for reuse by the the running program. An object is considered *garbage* (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a "dangling pointer" into a deallocated object.

This paper is intended to be an introductory survey of garbage collectors for uniprocessors, especially those developed in the last decade. For a more thorough treatment of older techniques, see [Knu69, Coh81].

1.1 Motivation

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies. A routine operating on a data structure should not have to know what other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be deallocated explicitly, some module must be responsible for knowing when *other* modules are not interested in a particular object.

[1] We use the term object loosely, to include any kind of structured data record, such as Pascal records or C structs, as well as full-fledged objects with encapsulation and inheritance, in the sense of object-oriented programming.

P. R. Wilson, "Uniprocessor garbage collection techniques", Proceedings of the International Workshop on Memory Management, St. Malo, France, September 1992. DOI: 10.1007/BFb0017182

Basic Garbage Collection

- Mark-sweep
- Mark-compact
- Copying collectors

Garbage Collection

- Avoid problems of reference counting and complexity of compile-time ownership tracking via **garbage collection**
  - Explicitly trace through allocated objects, recording which are in use; dispose of unused objects
  - Moves garbage collection to be a separate phase of the program's execution, rather than an integrated part of an objects lifecycle
    - Operation of the program (the **mutator**) and the garbage collector is interleaved
- Many tracing garbage collection algorithms exist:
  - Basic garbage collectors
    - Mark-sweep collectors
    - Mark-compact collectors
    - Copying collectors
  - Generational garbage collectors

Mark-Sweep Collectors (1/4)

- Simplest automatic garbage collection scheme
- Two phase algorithm
  - Distinguish live objects from garbage (*mark*)
  - Reclaim the garbage (*sweep*)
  - Non-incremental: program is paused to perform collection when memory becomes tight

Mark-Sweep Collectors (2/4)

- The **marking phase**: distinguishing live objects
  - Determine the **root set** of objects, comprising:
    - Any global variables
    - Any variable allocated on the stack, in any existing stack frame
  - Traverse the object graph staring at the root set to find other reachable objects
    - Starting from the root set, follow pointers to other objects
    - Follow every pointer in every object to systematically find all reachable objects
    - May proceed either breadth-first or depth-first
    - A cycle of objects that reference each other, but are not reachable from the root set, will not be marked
  - Mark reachable objects as alive
    - Set a bit in the object header, or in some separate table of live objects, to indicate that the object is reachable
    - Stop traversal at previously seen objects to avoid looping forever

Mark-Sweep Collectors (3/4)

- The **sweep phase**: reclaiming objects that no longer live
  - Pass through entire heap once, examining each object for liveness
    - If marked as alive, keep the object
    - Otherwise, free the memory and reclaim the object's space

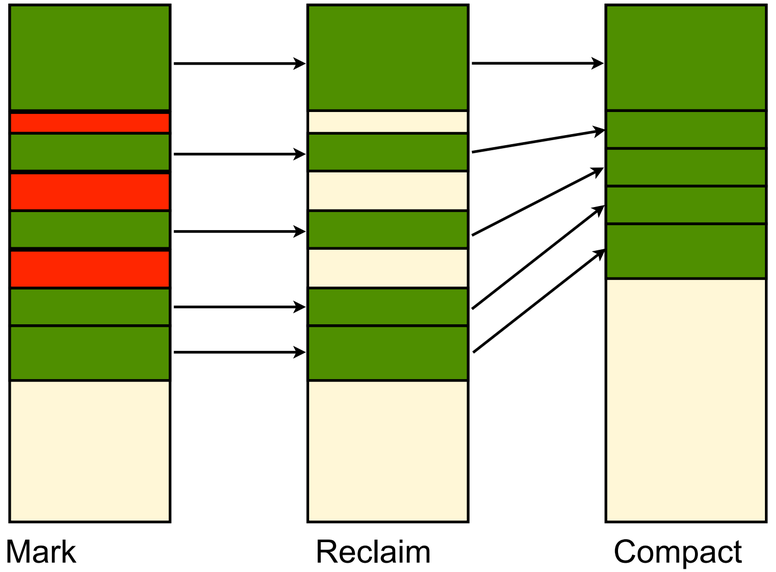
- When objects are reclaimed, their memory is marked as available
  - The system maintains a **free list** of blocks of unused memory
  - New objects are allocated in now unused memory if they fit; or in not-yet-used memory elsewhere on the heap
  - Fragmentation is a potential concern – but no worse than using **`malloc()`**/**`free()`**

Mark-Sweep Collectors (4/4)

- Mark-sweep collectors are simple, but inefficient:
  - Garbage collection is slow and has unpredictable duration
    - Program is stopped while the collector runs
    - Time to collect garbage is unpredictable, and depends on the number of live objects (time for the marking phase) and size of the heap (time to sweep up unused objects)
    - Unlike reference counting, mark-sweep garbage collection is slower if the program has lots of memory allocated
  - Garbage collection has no locality of reference
    - Passing through the entire heap in unpredictable order disrupts operation of cache and virtual memory subsystem
    - Objects located where they fit, rather than where maintains locality of reference
  - Fragmentation of free space is a concern
    - Since objects are not moved, space may become fragmented, making it difficult to allocate large objects even though space available overall

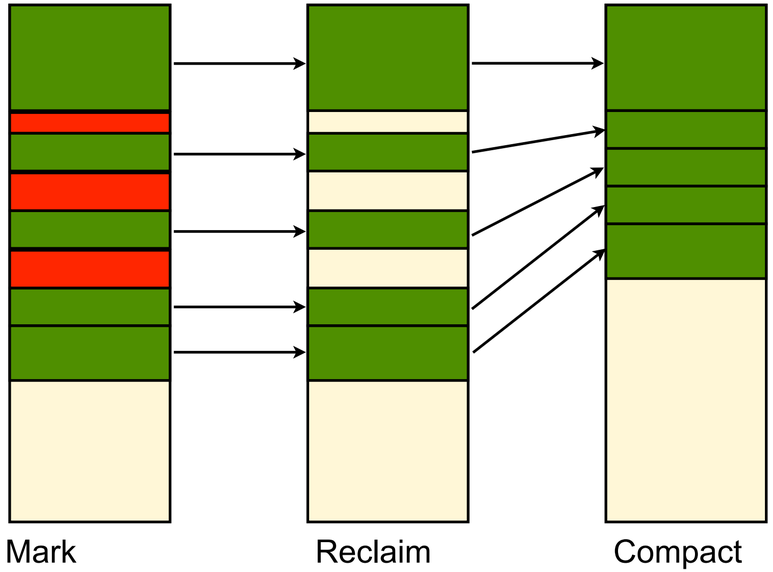
Mark-Compact Collectors (1/2)

- Goal: solve fragmentation problems and speed-up allocation, compared to mark-sweep collectors
- Three logical phases:
  - Traverse object graph, **mark** live objects
  - **Reclaim** unreachable objects
  - **Compact** live objects, moving them to leave contiguous free space
  - Reclaiming and compacting memory can be done in one pass, but still touches the entire address space



Mark-Compact Collectors (2/2)

- Advantages:
  - Solves fragmentation problems – all free space is in one contiguous block
  - Allocation is very fast – always allocating from the start of the free block, so allocation is just incrementing pointer to start of free space and returning previous value
- Disadvantages:
  - Collection is slow, due to moving objects in memory, and time taken is unpredictable
  - Collection has poor locality of reference
  - Collection is complex – needs to update all pointers to moved objects



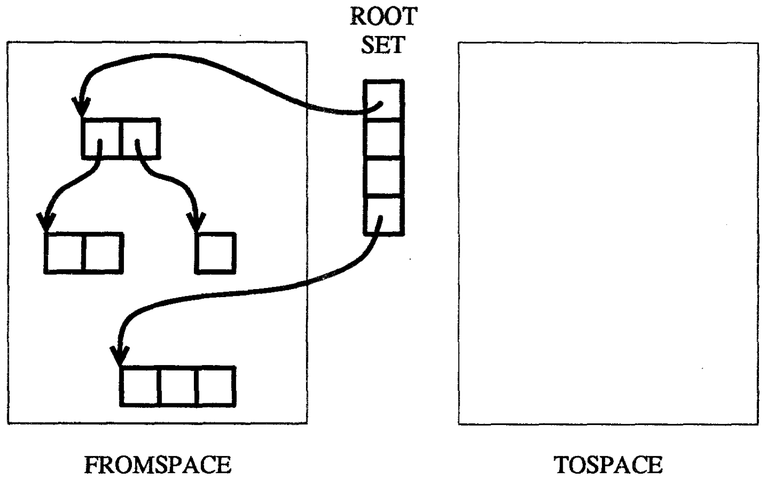
Copying Collectors (1/5)

- Copying collectors integrate traversal (marking) and copying phases into one pass
  - All the live data is copied into one region of memory
  - All the remaining memory contains garbage, or has not yet been used

- Similar to mark-compact, but more efficient
- Time taken to collect is proportional to the number of live objects

Copying Collectors (2/5)

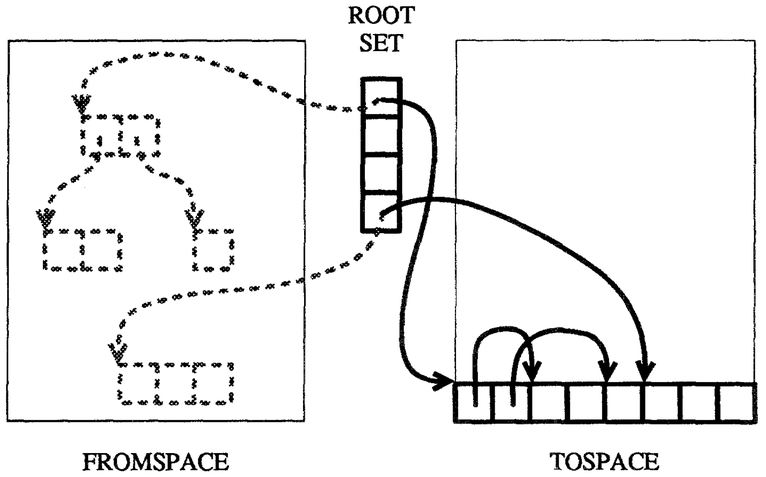
- Stop-and-copy using semispaces:
  - Divide the heap into two halves, each one a contiguous block of memory
  - Allocations made linearly from one half of the heap only
    - Memory is allocated contiguously, so allocation is fast (as in the mark-compact collector)
    - No problems with fragmentation when allocating data of different sizes
  - When an allocation is requested that won't fit into the active half of the heap, a collection is triggered



Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI 10.1007/BFb0017182

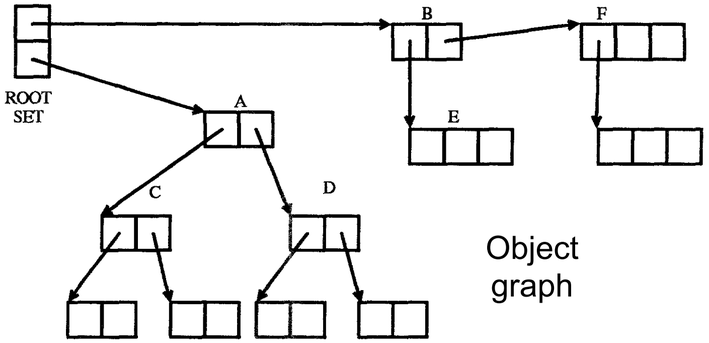
Copying Collectors (3/5)

- Stop-and-copy using semispaces:
  - Collection stops execution of the program
  - A pass is made through the active space, and all live objects are copied to the other half of the heap
    - Cheney algorithm is commonly used to make the copy in a single pass
    - Anything not copied is unreachable, and is simply ignored (and will eventually be overwritten by a later allocation phase)
  - The program is then restarted, using the other half of the heap as the active allocation region
- The role of the two parts of the heap (the two semispaces) reverses each time a collection is triggered

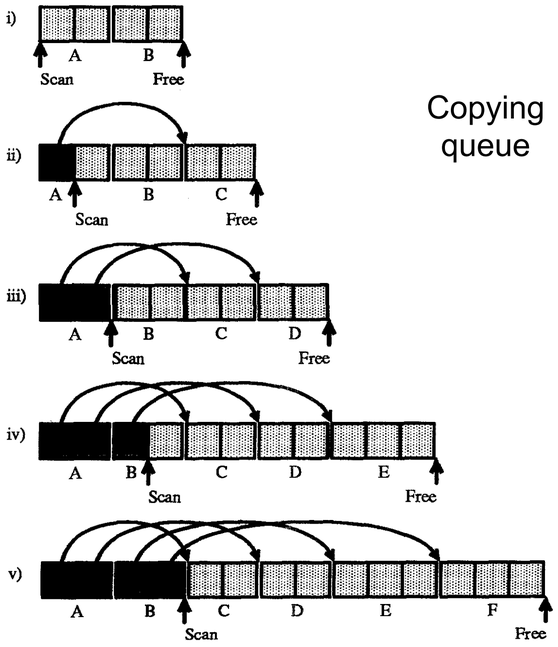


Source: P. Wilson, “Uniprocessor garbage collection techniques”, Proc IWMM’92, DOI 10.1007/BFb0017182

Copying Collectors (4/5)



Object graph



Copying queue

- The Cheney Algorithm: breadth-first copying
  - A queue is created, to hold the set of live objects to be copied
  - The **root set** of objects, comprising global variables and all stack allocated variables, is found and added to the queue
  - Objects in the queue are examined in turn:
    - Any unprocessed objects they reference are added to end of the queue
    - The object in the queue is then copied into the other semispace, and the original is marked as having been processed (pointers are updated as the copy is made)
  - Once the end of the queue is reached, all live objects have been copied

Source: P. Wilson, “Uniprocessor garbage collection techniques”, Proc IWMM’92, DOI 10.1007/BFb0017182

Copying Collectors (5/5)

- Efficiency of copying collectors:
  - Time taken for garbage collection depends on the amount of data copied, which depends on the number of live objects
  - Collection only happens when a semispace is full

- **If most objects die young**, can trade-off collection time *vs.* memory usage by increasing the size of the semispaces
  - A larger semispace takes longer to fill, so increases the how long objects need to live before they're copied
  - If most objects die young, most aren't copied
  - Uses more memory, but spends less time copying data

Summary: Basic Garbage Collection

- Mark-sweep, mark-compact, and copying collectors have similar cost:
  - **Differ in where the cost is paid**: at time of allocation or time of collection; in memory usage or in processing time

- The mark-compact and copying algorithms move data, so cannot be used with languages that cannot unambiguously identify pointers
  - Can’t move an object, if you can’t be sure all pointers to it have been updated

Basic Garbage Collection

- Mark-sweep
- Mark-compact
- Copying collectors