CSE 401/M501 – Compilers

Memory Management
and Garbage Collection

Hal Perkins – Guest Lecturer

Spring 2025

Final Announcements

- hw4 due last night; if you have late days remaining you can use them but get it in on time

- Please nominate great TAs for the Bandes award.
  - Both for CSE 401/M501 and for other courses

Final exam

- Tuesday 2:30-4:20 here (CSE2 G10)

- Closed book like midterm, but ok to have two 5x8 hand-written note cards (can keep midterm card or replace it)
  - Extra blank cards available end of class today

- Topic list and old exams posted on website now

- Review Q&A Monday 2:30, CSE2 G01 (not here)

- HW4 sample solutions available outside Gilbert’s office (CSE2 235) late Sunday or Monday
  - Copies of all hw solutions also available at Monday’s review session

References

- *Uniprocessor Garbage Collection Techniques*
  Wilson, IWMM 1992 (longish survey)

- *The Garbage Collection Handbook*
  Jones, Hosking, Moss, 2012 (book)

- Earlier version of this lecture by Vijay Menon, CSE 501; Jim Hogg, CSE 401/M501

Program Memory

- Typically divided into 3 regions:
  - Global / Static: fixed-size at compile time; exists throughout program lifetime
  - Stack / Automatic: per function, automatically allocated and released (local variables)
  - Heap / Dynamic: Explicitly allocated by programmer (malloc/new/cons)
    - Need to recover / recycle storage for reuse when no longer needed

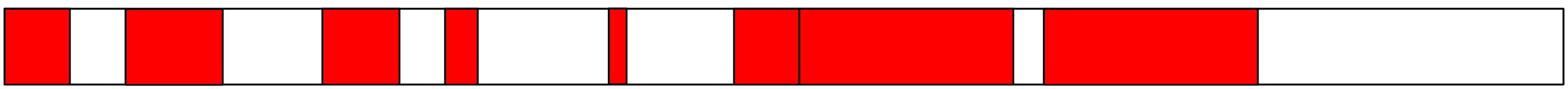
Manual Heap Management

- Programmer calls free/delete when done with storage
- Pro
  - Cheap
  - Precise
- Con
  - How do we enumerate the ways? the pain?
  - Buggy, huge debugging costs, …

Conventional Heap Storage

```
...
char* s = (char*) malloc(50);
...
free(s);
```

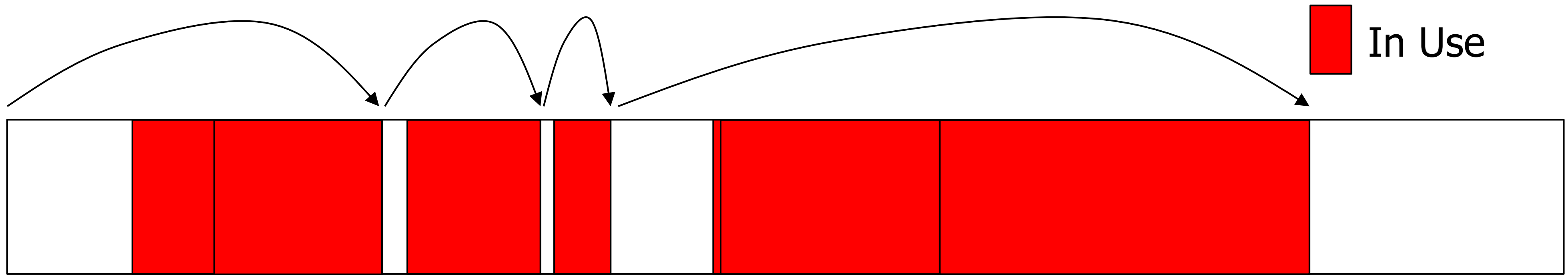
In Use



C Runtime Heap Memory

- Developer must remember to free memory when no longer required
- Eventual fragmentation => slow to malloc, slow to free

Heap Storage Fragmentation

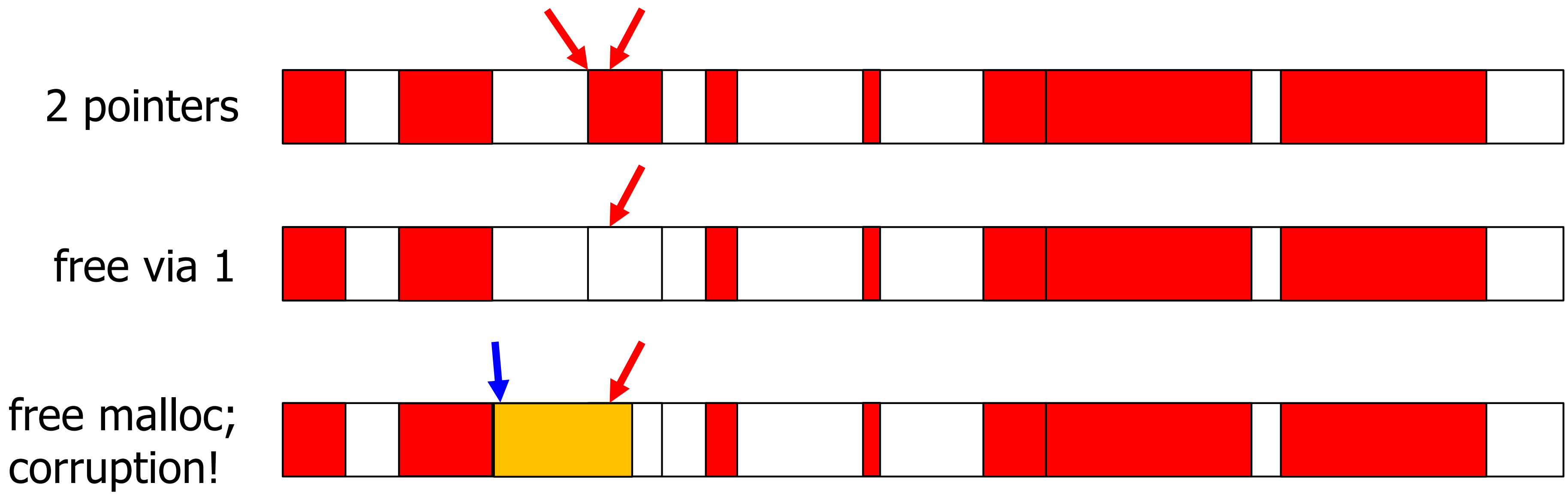


C Runtime Heap Memory

- malloc: walk the freelist to find a slot big enough for current request
- free: adjust freelist; collapse contiguous freespace
- fragmentation: plenty free chunks but none big enough for request
- cannot compact the used space - may contain pointers; may be pointed-at

Bugs

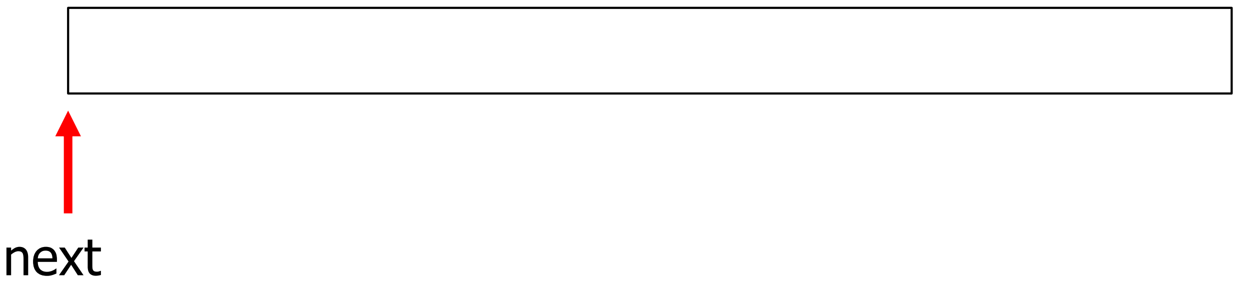
- Forget to free => eventually run out of memory
  - called a "memory leak”
- Call free, but continue to use!
  - called "use-after-free", or "dangling pointer"
  - memory corruption - wrong answers; crash if lucky!
  - major source of security issues
  - detect via "pool poisoning"

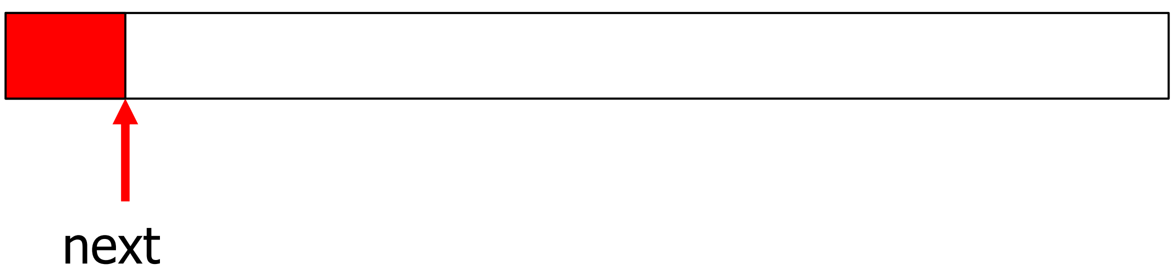


Garbage Collection

- Automatically reclaim heap memory no longer in use by the program
  - Simplify programming
  - Better modularity, concurrency
  - Avoids huge problems with dangling pointers
  - Almost required for type safety
  - But not a panacea – still need to watch for stale pointers, GC's version of "memory leaks"
    - i.e., pointers in live data to no-longer-used data

Garbage Collection

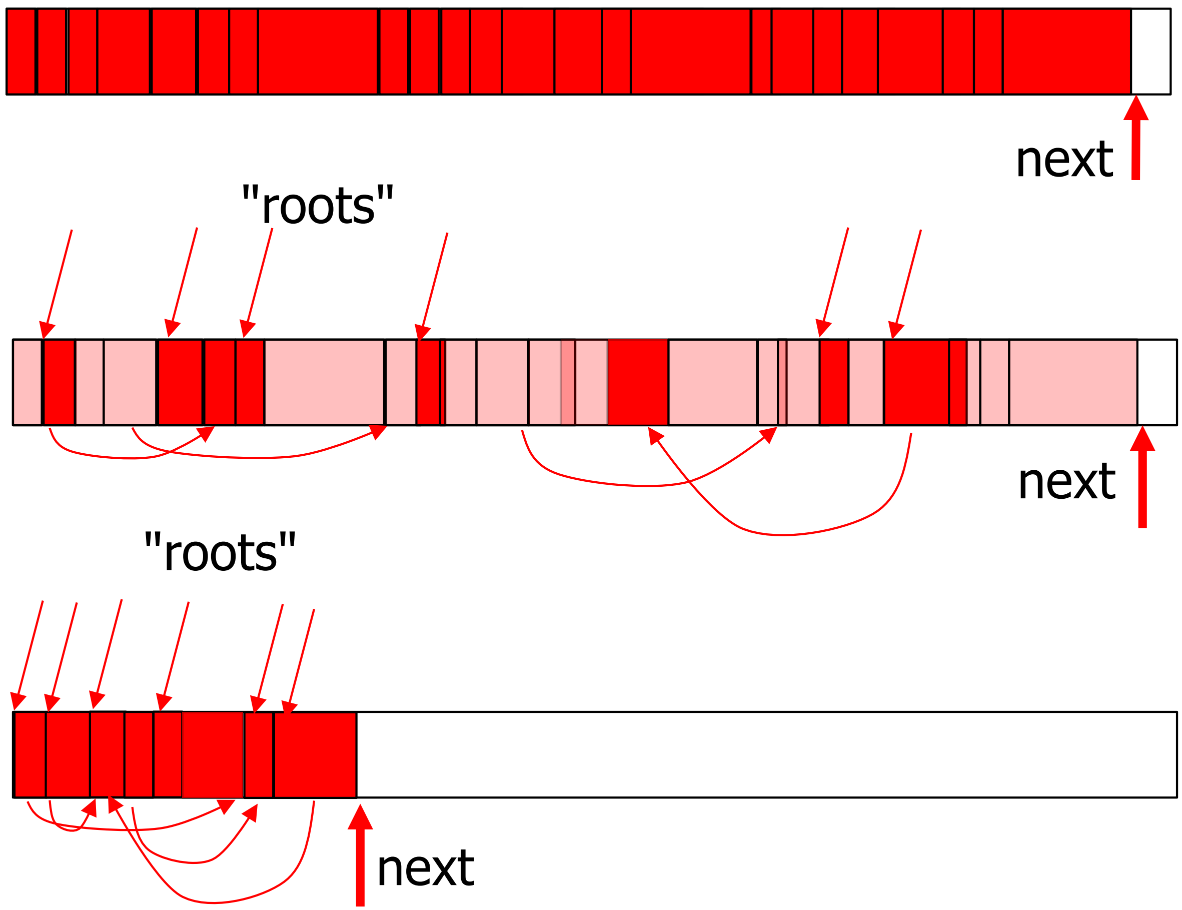




Allocate an object; fast!

Allocate more objects;
and one more, please?

Garbage Collection



Allocate another object

Trace reachable objects

Compact unreachables; update *all* pointers

GC does not find garbage: it finds live objects and ignores all other memory

Heap Characteristics

- Most objects are small (< 128 bytes)
- Object-oriented and functional code allocates a huge number of short-lived objects
- Want allocation, recycling to be fast and low overhead
  - Serious engineering required

Allocation

- Usually multiple free lists organized by size for small objects (8, 16, 24, 32, … depends on alignment); additional list for large blocks
  - Regular malloc does exactly the same
- Allocation
  - Grab a free object from the right free list
  - No more memory of the right size triggers a collection

What is Garbage?

- An object is *live* if it is still in use
- GC needs to be conservative
  - OK to keep memory no longer in use
  - Not ok to reclaim something that is live
- An object is *garbage* if it is not live

Reachability

- *Root set* : the set of global and local (stack + register) variables visible to active procedures
- Heap objects are *reachable* if:
  - They are directly accessible from the root set
  - They are accessible from another reachable heap object (pointers/references)
- Liveness implies reachability (conservative approximation)
- Not reachable implies garbage

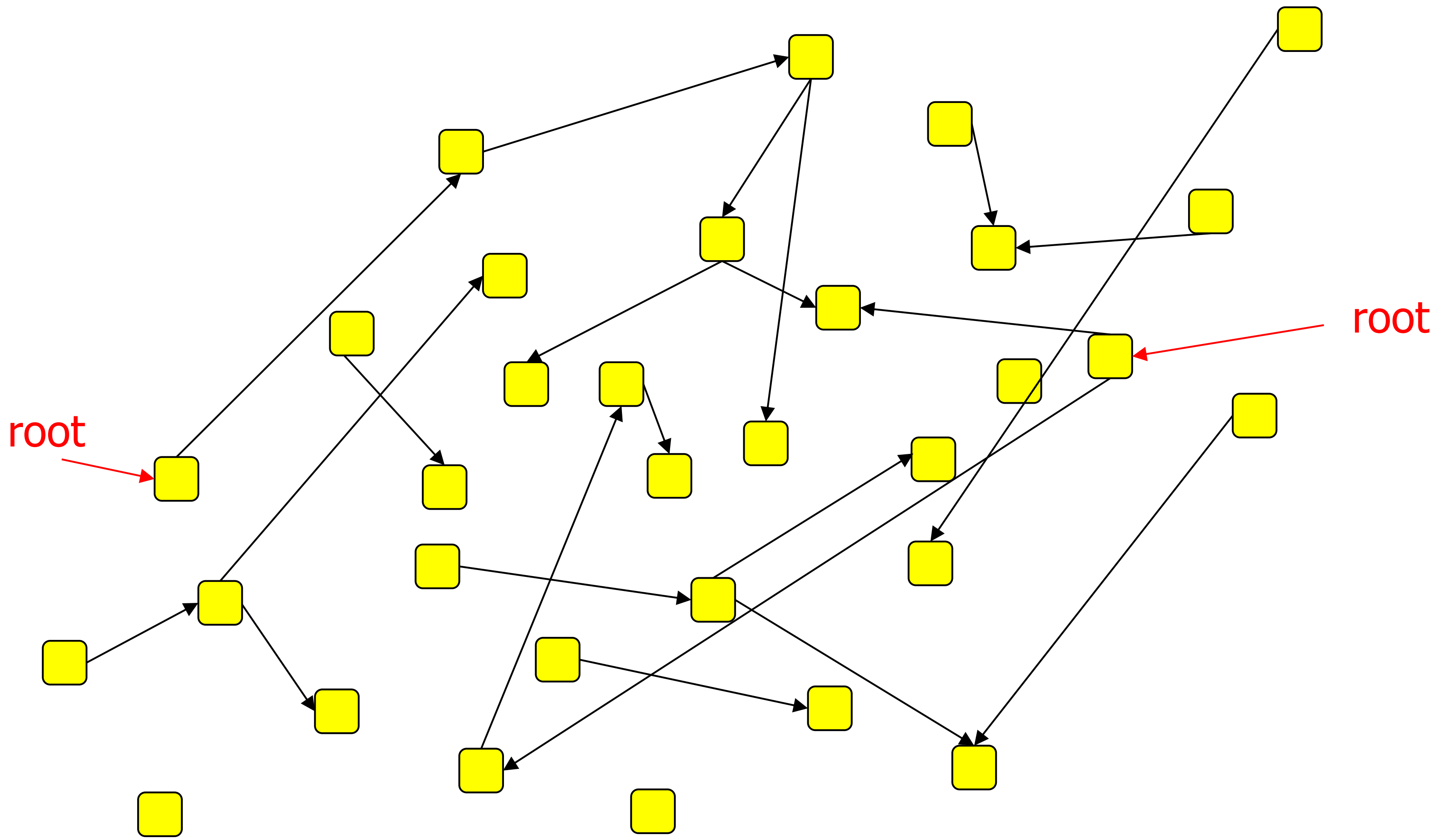
Tracing Collectors

- Mark the objects reachable from the root set, then perform a transitive closure to find all reachable objects
- All unmarked objects are dead and can be reclaimed
- Various algorithms: mark-sweep, copying, generational…

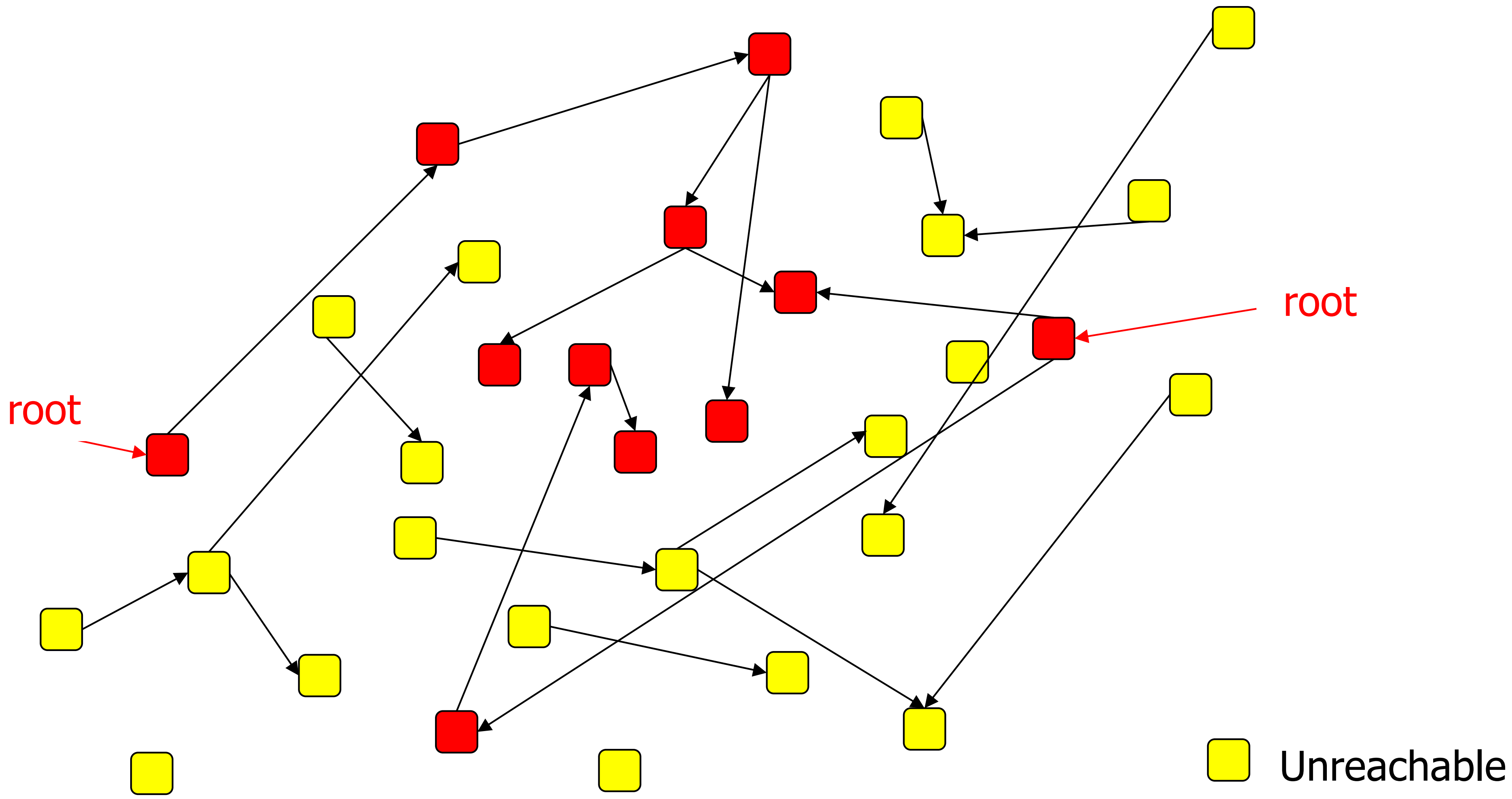
Mark-Sweep Collection

- Mark phase – find the live objects
  - Transitive closure from root set marking all live objects
- Sweep phase
  - Sweep memory for unmarked objects and return to appropriate free list(s)

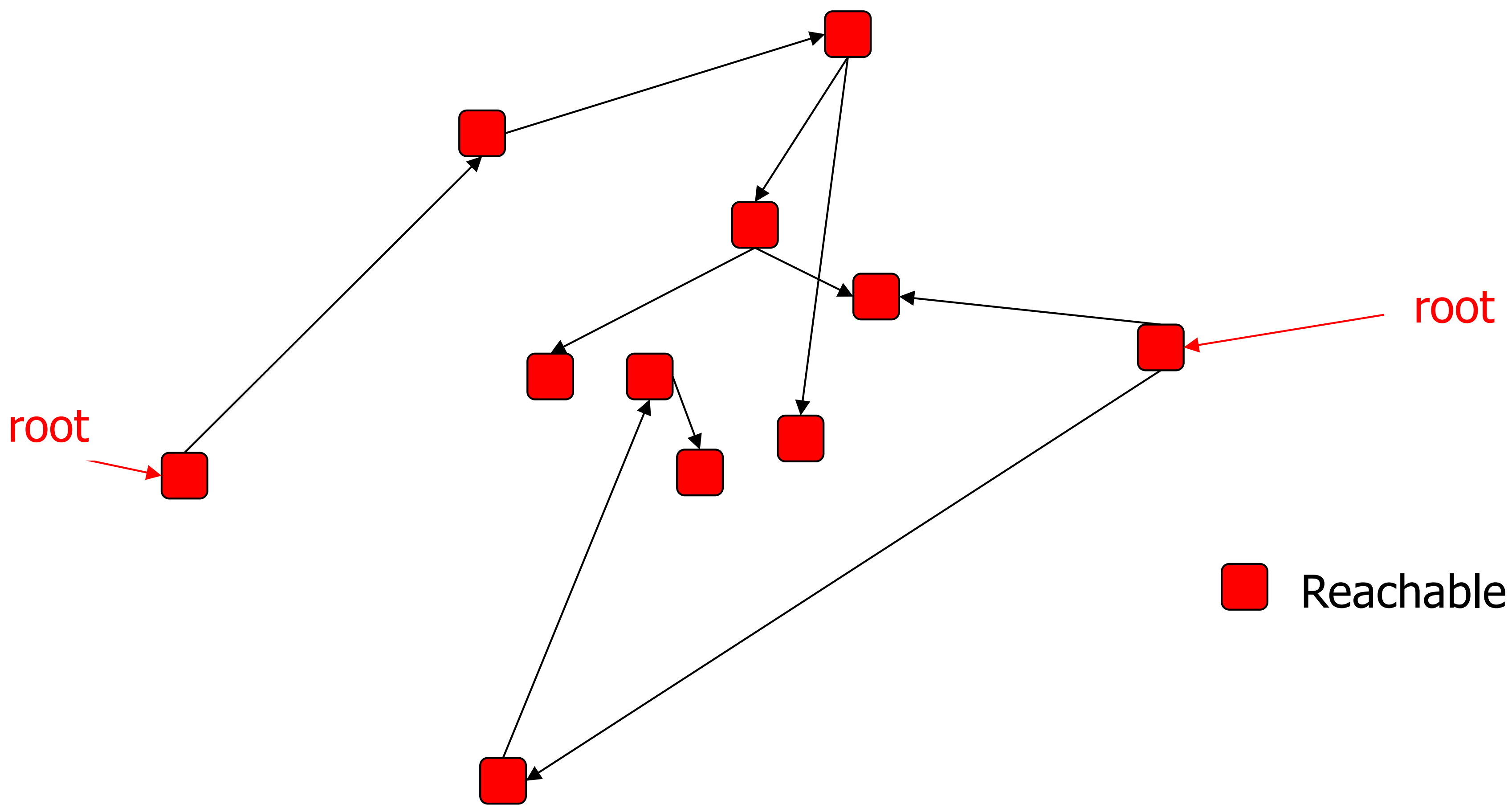
GC Start



GC Mark Phase



GC Sweep Phase



With memory free, now allocate space for object that provoked the GC

Reachability

- Compiler produces:
  - A *stack-map* at *GC safe points*
    - *Stack map:* enumerate global variables, stack variables, live registers (tricky stuff! Why?)
    - *GC safe points:* new(), method entry, method exit, back edges (thread switch points)
      - Stop all threads at one of their GC safe points and then ok to do a collection
  - *Type information blocks*
    - Identifies reference fields in objects (to trace the heap)

Mark-Sweep Evaluation

- Pro
  - Space efficiency
  - Incremental object reclamation
- Con
  - Relatively slower allocation time (free lists vs. “next chunk of heap”)
  - Can have poor locality of objects allocated at around the same time
  - Redundant work rescanning long-lived objects
  - “Stop the world I want to collect”

Semispace Copying Collector

- Idea: Divide memory in half
  - Storage allocated from one half of memory
  - When full, copy live objects from old half (“from space”) to unused half (“to space”) & swap semispaces
- Fast allocation – next chunk of to-space
- Requires copying collection of entire heap when collection needed

Semispace collection

- Same notion of root set and reachable as in mark-sweep collector
- Copy each object when first encountered
- Install forwarding pointers in from-space referring to new copy in to-space
- Transitive closure: follow pointers, copy, and update as it scans
- Reclaims entire “from space” in one shot
  – Swap from- and to-space when copy done

Semispace Copying Collector Evaluation

- Pro
  - Fast allocation
  - Locality of objects allocated at same time
  - Locality of objects connected by pointers (can use depth-first or other strategies during the mark-copy phase)
- Con
  - Wastes half of (virtual?) memory
    - Other copying/compacting collectors solve some of this
    - Be careful with VM – don't want compacting to thrash
  - Redundant work rescanning long-lived objects
  - "Stop the world I want to collect"

Generational Collectors

- Generational hypothesis: young objects die more quickly than older ones (Lieberman & Hewitt ‘83, Ungar ‘84)
- Most pointers are from younger to older objects (Appel ‘89, Zorn ‘90)
- So, organize heap into young and old regions, collect young space more often

Generational Collector

- Divide heap into two spaces: young, old
- Allocate new objects in young space
- When young space fills up, collect it and copy surviving objects to old space
  - Engineering: use write barriers to avoid having to scan all of old space on quick collections – most pointers that cross the boundary are from young objects to old
  - Refinement: require objects to survive at least a few collections before copying
- When old space fills, collect both
- Often use multiple generations, not just two

GC Tradeoffs

- Performance
  - Mark-sweep often faster than semispace
  - Generational better than both
- Mutator (i.e., user program) performance
  - Semispace is often fastest
  - Generational is better than mark-sweep
- Overall: generational is a good balance
- But: we still “stop the world” to collect

Advanced GC and Research Areas

- Parallel/concurrent garbage collection
  - Found in more production collectors these days
    - Tricky stuff – can't debug it into correctness – there be theorems here
- Locality issues
  - Object collocation
  - GC-time analysis
- Distributed GC

Compiler & Runtime Support

- GC tightly coupled with safe runtime (e.g., Java, CLR (C#, …), functional languages)
  - Total knowledge of pointers (type safety)
  - Tagged objects with type information
  - Compiler maps for information
  - Objects can be moved; forwarding pointers

What about unsafe languages? (e.g., C/C++)

- Boehm/Weiser collector: GC still possible *without* compiler/runtime cooperation(!)
  - New versions of malloc (& free) + GC to manage heap
  - If it looks like a pointer, it's a pointer
  - Mark-sweep only – GC doesn't move anything
  - Allows GC in C/C++ but constraints on pointer bit-twiddling
  - Surprisingly effective, particularly if program uses pointers as in a type-safe language (e.g., no pointer mangling, no (void*)int tricks, etc.)

Boehm/Weiser Collector

- Useful for development/debugging
  - Less burden on compiler/runtime implementor
- Used in various Java and .net prototypes, research implementations, production code if sufficiently effective
- Similar ideas for various tools to detect memory leaks, etc.

A bit of perspective…

- Automatic GC has been around since LISP I in 1958 (!)
- Ubiquitous in functional and object-oriented programming communities for decades
- Mainstream since Java(?) (mid-90s)
- Now conventional wisdom?