Incremental and Concurrent GC

GC for interactive and real-time systems

Interactive or real-time app concerns

- Reducing length of garbage collection pause
- Demands guarantees for worst case performance
- Generational GC works if:
  - Young generation is relatively small
  - Survival rate of objects is sufficiently low

Interactive or real-time app concerns

- Generational GC does not work if
  - Does frequent major collections
  - Survival rate is high
  - Attempts to improve expected pause time
    - At expense of worst case

A turn to incremental techniques

- Simplest is RCGC
  - Naturally incremental for all operations except
  - Deletion of last ptr to an object
    - However, recursive freeing can be circumvented

Hardware focus

- Sequential architectures
  - A task runs to completion on single processor
- Shared memory multiprocessors
  - May need locks on certain resources or parts of code
  - Synchronization very expensive

Incrementalizing sequential GC

- Interleave GC with mutator
- GC must make sufficient progress
  - Else mutator will run out of memory
  - Tune GC rate with rate of memory consumption:
    - Do a small amount of marking/copying with each allocation
    - To prevent mutator from running out of memory additional headroom is needed

Tuning GC rate

- To prevent mutator from running out of memory additional headroom is needed
  - R = # active words in heap at start of collection
  - K = # words traced at each allocation
  - Need R/K calls to allocator to mark all active words
  - At most R(1 + 1/K) live objects at end of tracing
    - Memory needed to prevent mutator starvation

Does incremental mean real-time

- Hard real-time: Results must be computed on time
  - Late result is as useless as incorrect result
  - Require worst-case guarantees NOT average case
  - Most demand space bounds, avoid virtual memory
- Soft real-time: Prefers that results are on time
  - Late results better than no results at all
- Many so called real-time GC cannot realistically meet worst-case deadline for hard real-time systems
  - They offer average case pause times

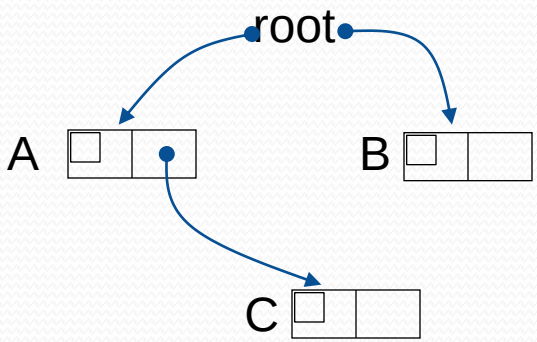
The need for synchronization

- Asynchronous execution
  - Independent execution of two or more process in a multitasking system
  - Execution of a process "in the background".
  - Other processes may be started before an asynchronous process has finished
  - Garbage collector
    - Can run as its own process (thread)
    - Can have fixed interleaving within single sequential process
    - **Introduces inconsistency in status of heap objects**

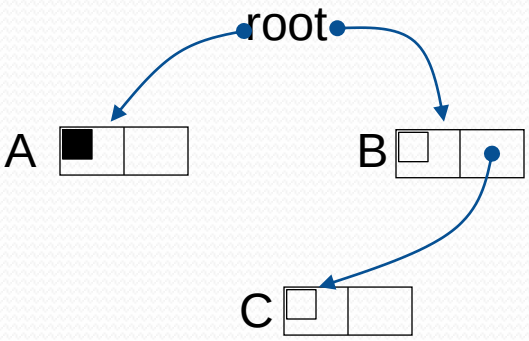
Synchronization defined

- Timekeeping which requires the coordination of events to operate a system in unison
  - System must be in sync
- The coordination of simultaneous threads or processes to complete a task
  - In order to obtain correct runtime order and avoid unexpected race conditions

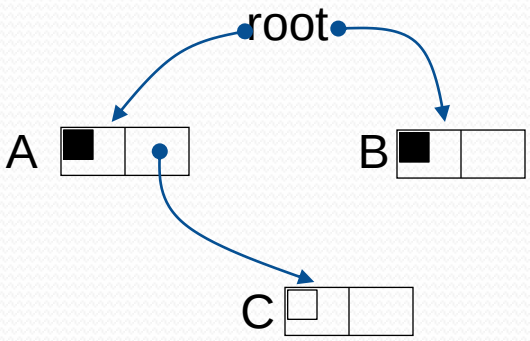
Synchronization example



update(right(B), right(A))
right(A) = NULL *step 1*
update(right(A), right(B))
right(B) = NULL *step 2*



Step 1



Step 2

Coherence problem

- Incremental mark-sweep collector
  - Poses multiple-readers, single writer coherence problem
  - Both mutator and collector read pointers
  - Only mutator modify (write) pointers
- Incremental copy collector
  - Poses multiple-readers, multiple-writers problem
  - Collector also writes pointers when it moves objects
- Mutator's view of the world **must** be consistent
- Collector has conservative view of reachability graph
  - Treat some unreachable objects as if they are reachable

Conservative collection

- GC and mutator do not need to have same view of reachability graph
- Introduces floating garbage
  - Became unreachable during last GC cycle
  - Will be collected during next cycle
  - Fragments heap
  - Increases effective residency of program
  - Puts pressure on GC

Judging incremental collectors

- Degree of conservatism is one parameter
- Bounds on mutator pause
  - GC must delay computation briefly at each step **to make progress**
  - Contain **uninterruptible** sections (increments)
    - Process root set
    - Check for termination of GC cycle

Tricolor abstraction

- Used for copying collection
- Used with Lin's algorithm (4 colors) for reclaiming cycles
- Was originally introduced by Dijkstra to describe incremental collection
  - Black
  - Grey
  - White
- GC cycle terminates when all reachable objects are colored black

Tricolor marking

Black:

- Object and immediate descendents have been visited
- GC finished with these objects; don't have to visit them again

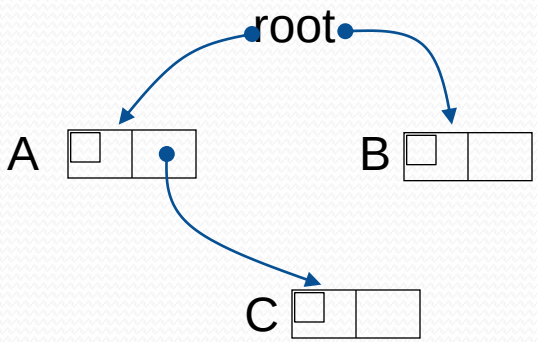
White:

- Objects not visited, are garbage **at end of tracing phase**

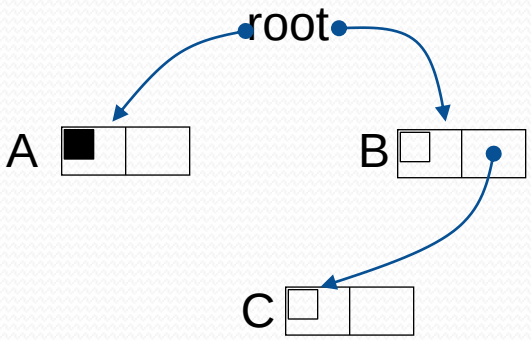
Gray:

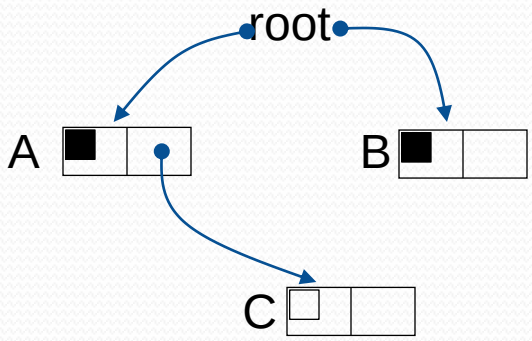
- Object must be visited by collector
- Object visited by collector but **descendants have not been scanned**
- **Object whose connectivity to rest of graph has been altered by mutator behind collector's back**

Synchronization example



update(right(B), right(A))
right(A) = NULL *step 1*
update(right(A), right(B))
right(B) = NULL *step 2*





Step 1 **Nodes A or C should be colored grey.** *Step 2*

Using barriers

- Mutator can disrupt GC by writing to black objects
  - Introducing black to white pointers
- How to prevent this?
  - Ensure mutator never sees white objects
    - Need to protect white objects with read barrier
  - Record where mutator writes pointers to white objects in black objects
    - Protect objects concerned with write barrier (mark them grey)
    - Collector re/visits objects concerned

Falsely reclaiming objects

- Object must be visible to mutator but invisible to collector
- Both of these conditions for failure must hold (during marking phase) for this to happen
  - A pointer to a white object must be written in a black object
    - This must be the only ref to white object
  - Original ref to white object must be destroyed
- Write barrier methods solve mutator-collector communication problem
  - tackle at least 1 condition of failure