Algorithms for Dynamic Memory Management (236780)

Lecture 5

Lecturer: Erez Petrank
Topics last week

- Generational Garbage Collection.
- The Train Algorithm (for the old generation).
Today: Concurrent Garbage Collection
Plan (tentative)

- On the fly collectors:
  - [Dijkstra-Lamport-Martin-Scholten-Steffens 1977]
  - [Doligez-Gonthier-Leroy 1993-94]

- Snapshot: the copy-on-write concurrent collector
  - [Demers-Weiser-Hayes-Boehm-Bobrow-Shenker 1990] + [Furusou-Matsuoka-Yonezawa 1991]

- Mostly concurrent collection

- Sliding Views:
Platform in Mind

- Multiprocessors (SMP) and multicores. Yesterday’s servers, today’s desktops, laptops, and smartphones.
- Shared memory.

![Diagram of CPU/Core, Cache, Memory]
Terminology

Informal Pause times

Stop-the-World

Parallel

Concurrent

On-the-Fly

Throughput Loss: 10%

© Erez Petrank
Part I: On-the-fly Garbage Collection

[Dijkstra-Martin-Steppens-Lamport-Scholten 1976]
Recall Three-Color Abstraction

- Black – objects that have been marked and their children have been marked as well.
- Gray – objects have been marked but their children have not been traced yet.
- White – objects that have not yet been marked.
An Intermediate Tracing View
Simplified Setting: A Graph

• Assume all objects are of fixed size and are represented as nodes in a directed graph.

• Possible manipulations: add a new edge, delete an edge, redirect an edge

• Simplifying assumptions:
  • Free-list holds all nodes ready for allocation.
  • A special nil node. All null edges point to the NIL node.

• Thus, no adds & deletes, only:
  • redirect edge &
  • get/return object to the free-list.
Reachable Nodes

- Several nodes in graph are defined roots.
- Other nodes are reachable (or not) from the roots.
- Note: program redirects an edge only between reachable nodes (source & target)
- Final abstraction: define a special root for the free-list, and NIL as a root.
- Now all possible program operations are redirecting an outgoing edge of one reachable node to point to another reachable node
Abstract Graph
Goals

• Find unreachable nodes concurrently with program run.
• Fine-grained synchronization between program and collector (short pauses).
• Low mutator overhead (low penalty on throughput).
• Correctness:
  • Do not reclaim reachable objects.
  • Some floating garbage allowed.
Terminology

- **Mutator** – program thread.
- **Collector** – the thread running the garbage collection.
- **Shading** – Changing white node to gray (no change for black or gray nodes).
- **Floating Garbage** - unreachable objects that were not reclaimed.

- Dijkstra et al.:
  - Only one mutator is allowed.
  - The collector employs a mark & sweep algorithm.
Basic Collector Algorithm

• Mark Phase
  • Mark the roots gray.
  • For every gray node: shade its successors, and then mark it black.
  • Quit when there are no gray nodes.

• Sweep phase: Visit each node once:
  • If it is white – append it to the free-list.
  • If it is black – color it white.

• Mutator cooperation:
  • When changing an edge, shade the new target.
Basic Collector Algorithm - Example
Basic Marking Algorithm

• Termination by monotonicity:
  • Nodes only grow darker during marking phase.

• Correctness invariant:
  • No black to white edge during marking phase.

• Problem: invariant assumes that pointer modification operation is atomic:
  • Change an edge and shade the new target.
Basic Marking Algorithm - Problem

- We do not want to use a critical section in the write barrier, so we must break it into two atomic writes. Either:
  - Option 1: Change and then shade, or
  - Option 2: Shade and then change.
- But:
  - Change and then shade violates the invariant.
  - Shade and then change: a concurrent sweep can erase the shade...
The Plan

- Use “change and then shade”, but employ a better analysis.
- The “no black to white” invariant is not preserved.
- But a finer invariant does, actually, two invariants.
The Invariants

• Two invariants ($P_1$ and $P_2$) hold simultaneously during the marking phase (after marking the roots):

• $P_1$: For any white reachable node $v$, there exists a "propagation path" starting at a gray node, continuing through white nodes and ending in $v$.

• $P_2$: A black to white edge is allowed once in the graph: the edge most recently written by the mutator.
Observation

- If there are no black to white edges, after marking the roots, the two invariants trivially hold.
  1. \( P_1 \) – For any white reachable node \( v \), there exists a “propagation path” starting at a gray node, continuing through white nodes and ending in \( v \).
  2. \( P_2 \) – A black to white edge is allowed once in the graph: the edge most recently written by the mutator.
The invariants depicted
Remarks:

- The statement of P2 assumes that there is only one mutator. (There exists the last modification operation.)
- P1 suffices for the proof.
  - At the end of marking there are no gray nodes. Thus, there is no reachable white node.
- But P₁ is too weak to be shown as invariant --> P₂ is needed for the induction.
Proof Base

• After root scanning P1 and P2 hold.
  • No black to white edge
  • Everything reachable is reachable from a gray node by a path of white nodes.

• Next we show that each action preserves P1 & P2.
• For simplicity and w.l.o.g. actions do not happen simultaneously.
Mutator Actions

• Mutator actions
  • Mutator actions sequence look like:
    - Change edge $e_1$
    - Shade new target of edge $e_1$
    - Change edge $e_2$
    - Shade new target of edge $e_2$
    - Change edge $e_3$
    - Shade new target of edge $e_3$
  • The interesting points in the sequence are after change and before shade.
    At other times, there are no black-white edges.
We need to show that mutator’s changing an edge preserves the invariant “$P_1$ and $P_2$”.

Preserving $P_2$ is easy.

- After “change $e_i$”, $e_i$ may become a black to white edge. It is the recently modified edge.
- After “shade target” $e_i$ is not a black-white edge anymore. At this time there are no black-white edges at all.
- By induction, $P_2$ is preserved.
Mutator Preserves $P_1$

- We show that $P_1$ holds, given that $P_2$ holds.
- Shading a vertex cannot destroy any propagation path.
- Thus, we concentrate on the “change” operation.
- We separate the analysis into two cases:
  - Source of modified edge is black
  - Source of modified edge is white or gray.
Preserving $P_1$ (black source)

- Suppose source of the modified edge is black.

- We need to show that we did not lose a propagation path to a reachable white object.

- **Preserving propagation paths:** the edge that “disappeared” could not have been part of a propagation path since the source is black.

- Thus, all white reachable vertices must still have propagation paths.
Preserving $P_1$ (non-black source)

• Suppose source of the edge was white or gray: By P2, after the modification there are no black to white edges. (The recently modified edge has a gray/white source.) Thus, $P_1$ holds trivially!

• (This is enough for the proof but:) what happens if (the last) propagation path is eliminated to the “previous” child?

• Answer: if there are no other propagation paths than this child is not reachable anymore.
Preserving $P_1$ (non-black source)

- Conclusion: Mutator actions do not break the invariant relation.
To check that the collector preserves the invariants we need to take a look into the algorithmic details.
Marking phase: (M nodes, in array)

for each root j do “shade the root j”;
while (gray nodes exist)
{
    choose a gray node N;
    shade N’s children and make N black
}

© Erez Petrank
Collector Actions (Makring phase)

- The collector may
  - C1: Shade a single root
  - C2: Check the color of a node
  - C3: Shade the successors of a node and make the node black.
- Run atomically, none of c1, c2, c3 can break (P₁ and P₂): Propagation paths cannot be eliminated, black-to-white edge cannot be created.
Collector Actions (Makring phase)

- Problem: C3 is a compound operation!
  - C3: Shade the successors of a node and make the node black.

- Details:
  1. For each child of N
     1. C = next child of N
     2. shade C
  2. make N black
Collector Actions (Makring phase)

- Details:
  1. For each child of N
    1. C = next child of N
    2. shade C
  2. make N black

- 1-1 and 1-2 clearly preserve P1 and P2. Shading a node cannot destroy a propagation path and cannot create a black-to-white edge.

- Let’s check Operation 2…
Operation 2
(Making the node black after shading its children.)

• If mutator does not interfere all along:
  • All is well: Blackened node is not on a propagation path (P1) and black-to-white edge is not created (P2).
  • Otherwise, mutator has modified one of node’s pointers.
• If mutator pointer modification completed,
  • then all children are gray - no problem.
• If mutator concurrently redirects an edge from father to a new son, but hasn’t shaded it yet:
  • Black-to-white edge may be created, legitimately.
  • A propagation path cannot be eliminated since previous descendant is gray.
Conclusion of proof

- P1 and P2 are preserved.
- By P1: collection is safe.
- Does this algorithm work with multithreading?
Intuition of proof

• If a black-white pointer is created, then the new white child must be reachable from somewhere else. That route must contain a propagation path because it has no black-to-white edges.

• This heavily relies on the fact that there is only one black-to-white edge and no other thread foils its propagation path.
Multi-Threaded User Code

- The algorithm fails with multi-threaded programs (= many mutators)
  - (P1 and P2) can’t be kept invariant.
    - P2 is not even well defined.
  - Let’s look at a bad example.
Multi-Threaded Program – Example
Multi-Threaded Program – Example

Mutator 1 in red
Mutator 2 in green
Multi-Threaded Program – Example

Mutator 1 in red
Mutator 2 in green
Multi-Threaded Program – Example

Mutator 1 in red
Mutator 2 in green
Multi-Threaded Program – Example

Mutator 1 in red
Mutator 2 in green
Properties of Dijkstra’s Collector

- Concurrent with no synchronization.
- The algorithm is theoretical.
  - Marking the free list,
  - Write barrier on roots,
  - Unable to deal with multithreading.
- But, the ideas are innovative.
  - [Doligez-Leroy-Gonthier94] solved some of the issues.
  - [Domani et al. 00] made it practical and incorporated it into the IBM JVM.
Dijkstra, in Retrospect

• “Our exercise has not only been very instructive, but at times even humiliating, as we have fallen into nearly every logical trap possible.”

• “It has been surprisingly hard to find the published solution and justification. It was only too easy to design what looked -- sometimes even for weeks and to many people -- like a perfectly valid solution, until the effort to prove it to be correct revealed a (sometimes deep) bug.”
The DLG Algorithm

[Doligez-Leroy 1993] A concurrent, generational garbage collector for a multithreaded implementation of ML

[Doligez-Gonthier 1994] Portable, Unobtrusive Garbage Collection for Multiprocessor Systems


• Implementation for OCAML.
• Semi-generational, local heaps for immutable objects.
• We will concentrate on the main idea.
  • No local heaps, No semi-generations, No use of immutable objects
• We will not study the full algorithm and proof, but only point out some relevant problems and solutions.
Topics

• Eliminating free-list traversal.
• A race between allocation and sweep.
• Avoid repeated heap traversals.
• Using handshakes to accommodate “shade and then change”.
• Eliminating write-barrier on local variables.
Eliminating the free-list scan

- Dijkstra’s algorithm traverses the free-list. This does not make sense in practice.
- DLG added a fourth color:
  - Blue = free list (neither traced, nor reclaimed)
  - White = unmarked
  - Gray = marked, children not visited
  - Black = marked and children visited.
The Color of New Objects

• If we allocate white – it may be reclaimed.
• During mark
  • black – it is true that they are reachable and their sons (none exist) have been traversed.
  • (gray is also OK, but no termination guarantee.)
• During sweep
  • Depending on object location.
  • white, if already swept
  • gray otherwise - to avoid reclamation
Race Allocation - Sweep

Allocation:
- If phase = marking then
  - Set object to black
- Else
  - If address(object) < sweep_pointer then
    - Set object to white
  - Else
    - Set object to black

Sweep:
- If object(sweep_pointer) is black set to white
- If object(sweep_pointer) is white reclaim.

Two problems:
- Perhaps phase value is outdated
- Perhaps sweep_pointer is outdated
The Solution

• It is always safe to set an object to gray
  • Allocation:

  1. if phase = marking then
  2. set the object to black;
  3. if phase = sweeping then
  4. set the object to gray;
  5. else
  6. if address(object) < sweep_pointer then
  7. set the object to white;
  8. else
  9. set the object to gray;

• Sweep:
  • If object(sweep_pointer) is black: set to white
  • If object(sweep_pointer) is white: reclaim.
Avoid Repeated Heap Traversals

- Marking terminates when there are no gray objects in the heap.
- Saving heap traversals:
  - DLG went over the heap to find gray objects.
  - The practical Java implementations used a markstack
  - Parallel access to a markstack must be addressed in a modern implementation. We will elaborate in future lectures.
Recall Basic Write Barrier Problem

- Write barrier with atomic actions. Either:
  - Option 1: Change and then shade, or
  - Option 2: Shade and then change.
- But:
  - Change and then shade violates the invariant.
  - Shade and then change can violate that relation if there was sweeping phase in the middle.
- Dijkstra used option (1) with the extra P2. But P2 is not relevant for multithreading.
Write Barrier & Coordination

- DLG use the other option: “Shade and then change”
- Problem: shading may disappear before the change.
  - sweep terminates & marking starts again.
The Problem

- A long write extends between two collections.

Solution: before starting a new mark, make sure each thread completes a write.
Write Barrier & Coordination

- DLG use the other option: “Shade and then change”
- Problem: shading may disappear before the change.
  - sweep terminates & marking starts again.
- Such a scenario is prevented by coordinating a new collection with the mutators.
- Before starting a new collection the collector makes sure each mutator is not in the middle of a write.
- Simple option: stop all threads before starting the collection, make sure none is in the middle of a write, and start the collection.
No Thread is Continuously Writing

Timing 1
- Sweep
- Mark

Timing 2
- Sweep

Timing 3
- Sweep
- Mark
Handshakes

- Stopping all threads and making sure they are all “in a good state” is costly and unnecessary.
- We only need to know that each of them is not stuck in a write.
- We can check them one-by-one using a handshake.
- **Handshake:**
  - Collector tells mutators that it has started tracing by raising a flag.
  - Each mutator responds by raising a local flag.
  - Response does not happen during a write!
  - Handshake ends when all mutators respond.
Write-Barrier on Local Variables

- The majority of program updates are executed on the local variables (the roots).
- Modern collectors avoid a write-barrier on the roots.
- Can we avoid the write barrier and not miss live objects?
Eliminating stack write barrier

- With Dijkstra, target of modification was shaded.
- Can it work without a write barrier on the roots?
- Assume an atomic “shade and change” for this discussion. (Even that would not work…)
Concurrent modification (cont.)

- Shading target values (only for heap pointers)
A Second Try (no stack write barrier)

- Will it work better if we shaded the old value of the pointer (instead of the new target)?
Yes and No

- **If** we
  - **stop all threads**
  - Scan all roots while threads stopped
  - Initiate write barrier (shading old for any heap pointer update)
  - Resume threads
- **Then all is OK.**
  - Because anything reachable from the roots during the halt time must be blackened by the end of the concurrent trace.
Using Handshakes

- If we
  - stop one thread at a time (via a handshake)
  - Scan its roots while halted
  - Initiate its write barrier (shading old for any heap pointer update)
  - Resume the thread
- Then it doesn’t work.
A Problem with Using a Handshake

- T1 scans roots (B) & starts w.b.
- T1 grabs A
- T2 eliminates B -> A
- T2 scan roots and starts w.b.
A Problem with Using a Handshake

- T1 scans roots (B) & starts w.b.
- T1 grabs A
- T2 eliminates B -> A
- T2 scan roots and starts w.b.

Let’s use Two handshakes; still problems exist…
A Problem with Graying Old & 2 HS

- Assume two handshakes, and shading old value.
- There is a time in which a thread can temporarily store objects in the heap without the GC noticing.
- Consider shade and change of a (heap) pointer p.
- In between the two, if a parallel thread assigns a different object to p, and then removes it, the collector does not know about it.
- We use this “temporary un-noticed assignment” to hide a pointer from the garbage collector.
- Other thread puts object in temp place before it reports its roots, and gets it back afterwards.
The specific scenario

- (T1 running w.b.).
- T1 now plans to write B into A.ptr
  - MarkGray A.ptr (null)
- A.ptr = B

- T2 has a local pointer (lp) to C (has not reported roots yet)
  - T2 writes C to A.ptr
    - MarkGray A.ptr (null)
    - A.ptr = C
    - lp = null
  - T2 reports roots
    - lp <- A.ptr (so C is alive)

The collector is not aware of C being alive and reclaims it.
Another try: Shade New as Well

- The solution is to let T2 (and everybody else) also mark the object that it writes to the heap, in this case, C.
- Now C cannot be hidden at the time roots are reported.

- T2 has a local pointer (lp) to C (has not reported roots yet)
- T2 writes C to A.ptr
  - MarkGray A.ptr (null)
  - A.ptr = C
  - MarkGray C
  - lp = null
- T2 reports roots
  - lp = A.ptr (so C is alive)
Are we Done?

- Not really. To obtain correctness, we must use three handshakes.
At What Interval should we Mark New?

- 2 handshakes are not enough!
- Suppose we start graying old and new after the first handshake.
- We give roots in the second handshake.
- A counterexample follows. The idea is that one thread will store locally before running the first handshake. The other thread will make a long write that will allow the temporary store.
The specific scenario

- T1 goes into a h.s. 1
- T1 now plans to write B into A.ptr
  - MarkGray A.ptr (null)

- A.ptr ?? B
- T1 runs h.s. 2 & reports its rots.

- T2 has a local pointer (lp) to C
  - T2 writes lp to A.ptr (no write barrier yet)
  - lp ?? null
  - T2 runs h.s. 1
  - T2 reports roots (at h.s. 2)
  - lp <- A.ptr (so C is alive)

The collector is not aware of C being alive and reclaims it.
Solution: Add another h.s.

- The solution is to have an additional h.s.
- This way, it is not possible for T2 to store the local pointer anymore.
- T2 needs to store the temp pointer before it runs first h.s. (before it runs the barrier), and it must leave it there until it finishes reporting the roots.
- But if T2 runs three h.s. in between, then T1 must be making at least one h.s., thus, T1 cannot have a long write (to cover the temp) for all this time.
Handshakes

• To summarize, 3 handshakes are used:
  • Tell mutators to start the write-barrier
  • Tell mutators that root marking is approaching
  • Tell mutators to mark their roots (marking started)
• After responding to the first handshake and before responding to the third, the mutator’s write barrier marks “old” + “target”. During the rest of the marking phase, only “old” is marked.
• No real proofs provided…
Timing diagram for global variables
No Proofs, No Algorithm

- We will not fully present or partially prove the DLG collector.
- We only discussed major issues in the design.
- The proof that appears in the paper cannot be taught in (this) class.
- Producing a simpler proof = a project!
We will not prove DLG.
Proof in the paper is "computerized".
Producing a "human" proof = a project!

An excerpt from the proof:

\[
\begin{align*}
\text{We will not prove DLG.} \\
\text{Proof in the paper is "computerized".} \\
\text{Producing a "human" proof = a project!}
\end{align*}
\]
Properties

• On-the-fly.
• No write barrier on local variables.
• Adequate for multithreading.
• Handshakes & write-barriers overhead small.
• Made practical in a series of works.
• A modern version of the algorithm (adapted for Java) was/is used with the IBM production JVM on some IBM platforms.
Summary

• Dijkstra et al.:
  • Concurrent GC
  • Full proof (on abstraction)
  • But: theoretical, only one mutator.

• DLG:
  • On-the-fly
  • Full proof (less abstract)
  • Practical: several threads, no write barrier on roots, no scan of free-list.
  • Extension implemented on commercial products.