Garbage Collection in Lua

Roberto Ierusalimschy

Automatic Memory Management

- Releases memory automatically, when it is not needed anymore.
- Two main approaches:
  - reference count
  - garbage collection
- Lua uses garbage collection.
  - main reason: in dynamically-typed languages, reference count adds overhead even if a program never allocates memory

The Lua GC

- All objects in Lua are subject to garbage collection.
  - tables, functions, “modules”, threads (coroutines), etc.
- Only objects accessible from the *root set* are preserved.
  - root set: the registry and shared metatables.
  - the registry contains the global table (`_G`), the main thread, and `package.loaded`.

The Lua GC

- The collector operates on top of standard allocation functions.
- All objects linked in a long list.

Lua GC until Version 5.0

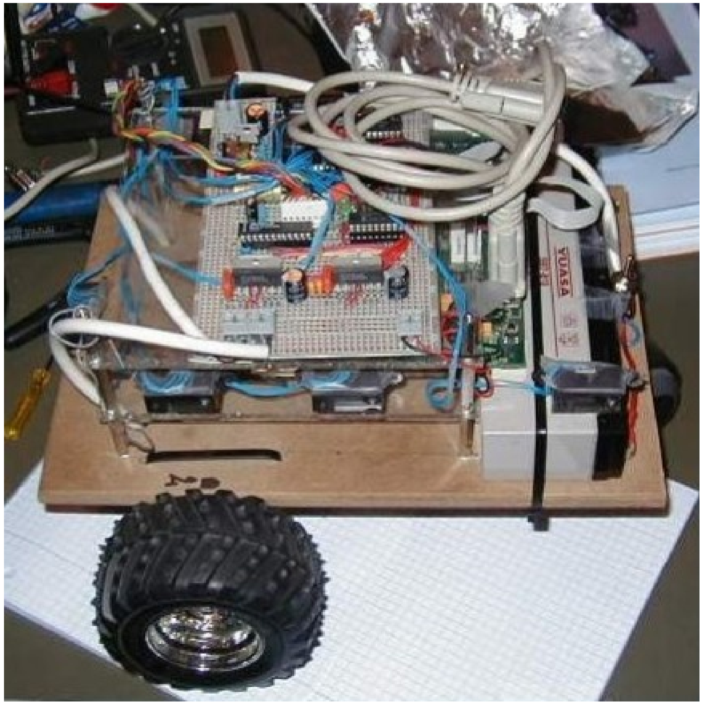
- Basic mark & sweep collector.
- Mark: traverse the object graph, starting from the root set, marking the live objects.
- Sweep: traverse the long list of all objects, deleting those not marked.
- Between mark and sweep, the collector separates and resurrects objects to be finalized.

The Collector's Pace

- The pace of a collector is a key component in system's performance.
- A collector that never runs has zero CPU cost, but a huge memory cost.
- A collector that runs all the time has zero memory overhead, but a huge CPU cost.
- The sweet spot is somewhere in between.
  - New collection when memory use is twice the use at the end of last collection.

Lua GC until Version 5.0

- Main drawback of Mark & Sweep: pauses in the program execution during a GC cycle can be very long.



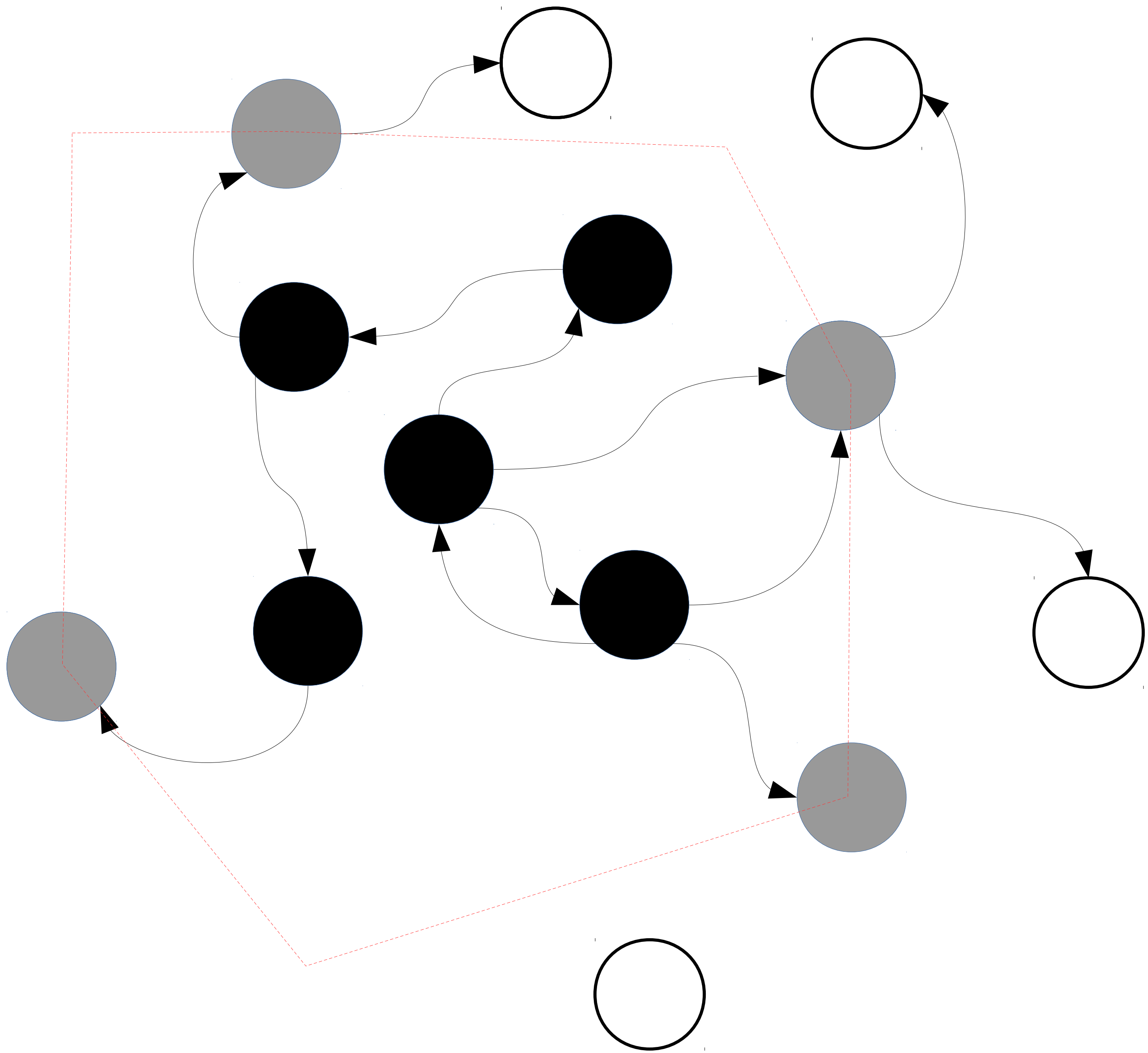
In version 5.1, Lua got an incremental collector. An *incremental* collector interleaves the execution of the collector with the main program.

The *Mutator*

- From the garbage collector's point of view, the program is just some nuisance changing the data it is trying to collect: the mutator!

Tri-color Collector

- Each object is in one of three states: white, gray, or black.
- Non-visited objects are marked white.
- Visited but not-traversed objects are marked gray.
- Traversed objects are marked black.



Invariants

- Objects in the *root set* are gray or black.
- A black object cannot point to a white object.
- Gray objects define the boundary between the black objects and the white objects.
- Collection advances by traversing gray objects, turning them black.
  - which may create new gray objects
- Collection ends when there are no more gray objects.

Barriers

- The mutator can break the invariant of black objects not pointing to white ones.

```
t.x = {}
```

- A *write barrier* tests this case and restores the invariant if necessary.
  - slows down all assignments
- It can either move forward the white object to gray or move backward the black object to gray.

Some Heuristics

- Objects moved back to gray are kept in a separate list to be traversed only in the atomic phase.
  - avoids a ping-pong
- Stacks are kept gray.
  - avoids barriers when writing to the stack!

Some Heuristics

- Assignment to tables moves a black table back to gray.

```
for i = 1, N do a[i] = <exp> end
```

- Assignment of a metatable moves a white metatable forward to gray.

```
setmetatable(obj, mt)
```

The Atomic Step

- The mark phase is ended by an *atomic step*.
- This step traverses all “gray again” objects.
  - including stacks
- It clears weak tables.
- It separates objects to be finalized, resurrecting them and their graphs.
- It clears weak tables (again?).

The Incremental Collector's Pace

- An incremental collector runs alternated with the mutator.
- At what pace?
- Most collectors measure “time” by memory allocation.
- How to translate bytes to GC activity?
  - the fallacy of “big userdata”

The Incremental Collector's Pace

- Two variables control the pace of the incremental collector.
- The *pause* controls by how much memory has to grow before starting a new cycle.
  - the fixed “2” in the old scheme.
- The *multiplier* controls the translation of bytes to GC work.
  - still somewhat mystifying

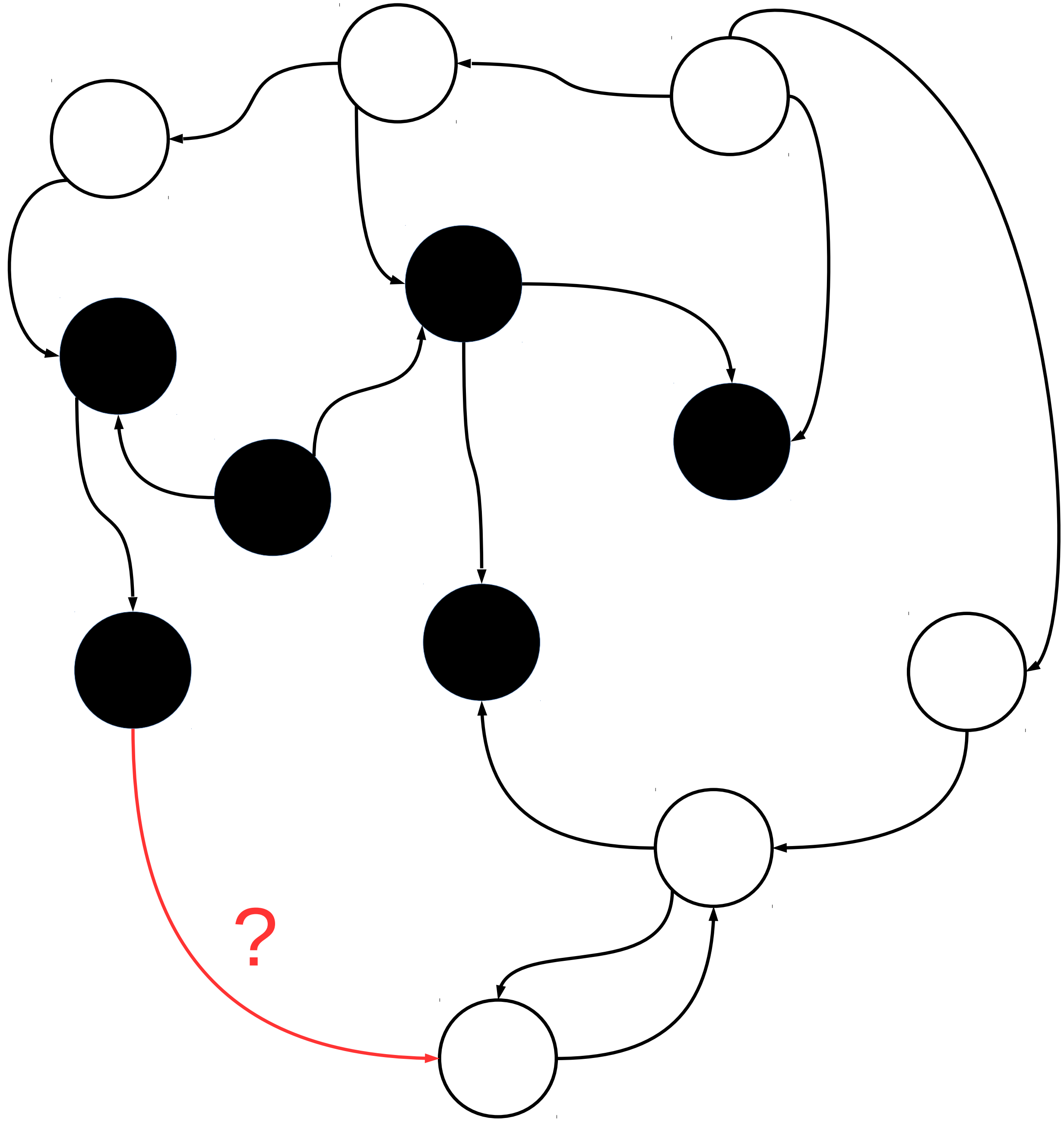
An incremental collector reduces the length of pauses, but it does not reduce the overall overhead of the collector; quite the opposite.

The Generational Collector

- The generational hypothesis: most objects die young.
- So, the collector could concentrate its efforts in young objects.
- All objects are classified as young or old. They are young when created; after surviving two collections, they become old.
- In a *minor* collection, the collector traverses and sweeps only young objects.

The Generational Invariant

- An old object should not point to a new one.
- Not so easy to keep as the incremental invariant.
- Both moving “forward” or “backward” have their problems.
  - forward: creates too many old objects (and breaks invariant)
  - backward: breaks invariant somewhere else



The *Touched* Objects

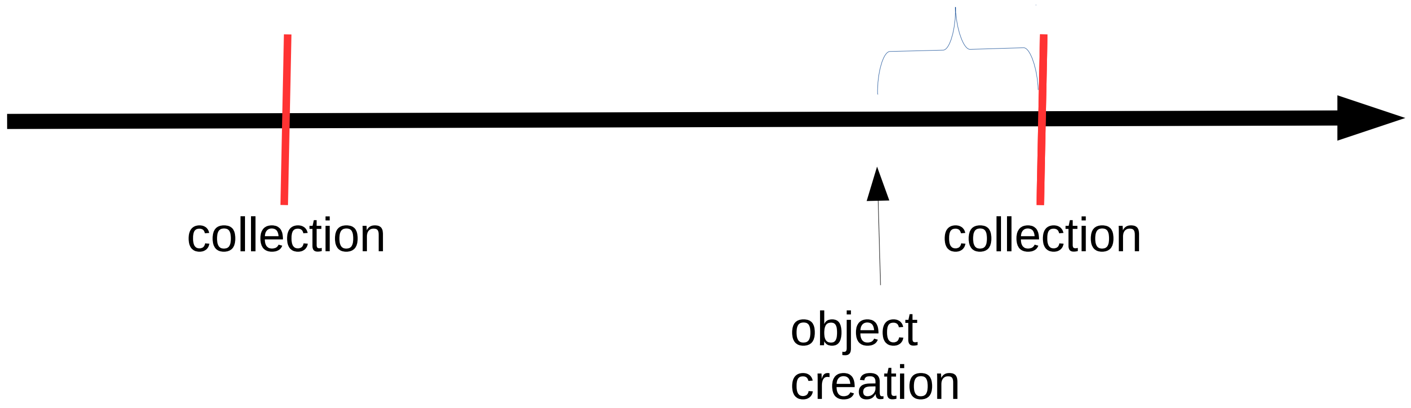
- If a back barrier detects an old object pointing to a new one, the old object is marked as *touched* and put in a special list.
  - not totally unlike the gray-again list
- Touched objects are also traversed (but not collected) in minor collections.
- After two cycles, a touched object goes back to regular old, unless it is touched again.

What was wrong with the generational collector in Lua 5.2?

What was wrong with the generational collector in Lua 5.2?

Objects had to survive one GC cycle before becoming old.

Surviving One Cycle

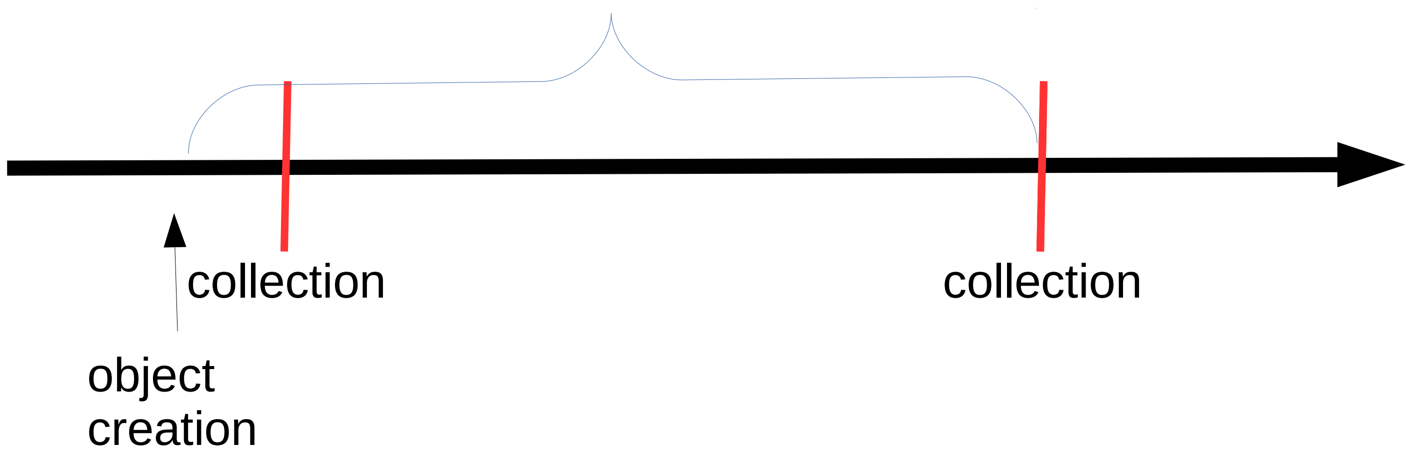


Object may survive an infinitely small interval before becoming old.

Surviving One Cycle

- Much simpler implementation.
- After a collection, all surviving objects become old, so the changes cannot break the invariant.
- List of touched objects can be erased.

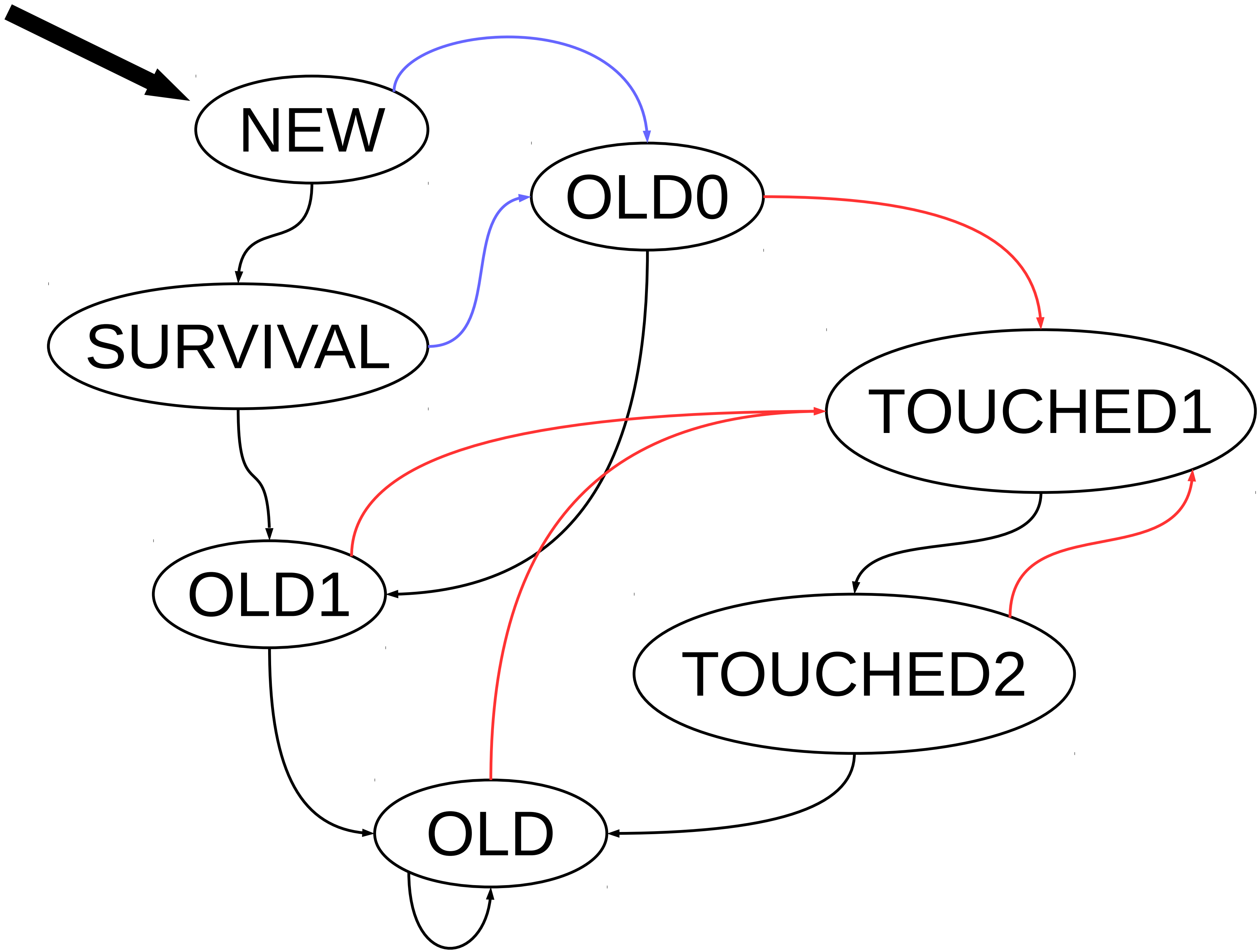
Surviving Two Cycles



Object must survive at least a full collection before becoming old.

Surviving Two Cycles

- At the end of a collection, some new objects become old, some do not.
  - breaks the invariant!
- List of touched objects must be corrected to next cycle.


NEW
OLD0
SURVIVAL
TOUCHED1
OLD1
TOUCHED2
OLD

Final Remarks

- When the generational hypothesis holds, a generational collector can reduce the overhead of the GC.
- Not always the hypothesis holds.
  - batch programs
- Testing a collector is hard.