# CS577 Sp'05 Lecture Notes
# Lecture 15



## Conservative Collection

Standard GC algorithms rely on precise identification of pointers.

This is hard in "uncooperative" environments, i.e., when the mutator (and its compiler) are not aware that GC will be performed. This is the normal case for C/C++ programs.

(Hence also an issue for portable Java implementations based on C, and for native functions.)

Basic problem: the mutator and collector can no longer communicate a root set.

(Also hard if mutator isn't sure about roots. E.g., for JVM stack must know statically which entries are pointers – easy except for **subroutines**.)

Idea: for any scanning collector to be correct, it's essential that every pointer be found. But for non-moving collectors, it's ok to mistake a non-pointer for a pointer – the worst that happens is that some garbage doesn't get collected.

**Conservative collectors** scan the entire register set and stack of the mutator, and **assume** that anything that **might** be a pointer really is a pointer.



## Issues in Conservative Collection

• Some bit patterns that are actually integers, reals, chars, etc. will be mistaken for pointers, so the "records" they "point" to will be treated as live data, causing **space leaks**.

• Accidental pointer identifications can be greatly decreased by careful tests, e.g., must be on a page known to be in the heap, at an appropriate alignment for objects on that page; data at "pointed-to" location must look like a heap header.

• Can further reduce false id's by not allocating on pages whose addresses correspond to data values known to be in use.

Major problems:

• Because they must filter large numbers of potential roots, conservative collectors tend to be slow.

• Collector must still be able to find all **potential** roots in registers and stack frames.

• Pointers must not be kept in "hidden" form by mutator code that does weird pointer arithmetic.

See widely-used **Boehm-Demers-Weiser collector**:

`http://www.hpl.hp.com/personal/Hans_Boehm/gc`



## Object Lifetimes

Major problem with tracing GC: long-lived data get traced (scanned and/or copied) repeatedly, without producing free space.

**(Weak) Generational Hypothesis:** "Most data die young."

I.e., most records become garbage a short time after they are allocated.

If we equate "age" of an object O is equated with amount of heap allocated since O was allocated, this says that most records become garbage after a small number of other records have been allocated.

Moreover, the longer an object stays live, the more likely it is to remain live in the future.

These are **empirical** properties of many (not necessarily all) languages/programs.

Implication : if you're looking for garbage, it's more likely to be found in recently-allocated data, e.g., in data allocated since the last garbage collection.

## Generational Collection

Idea: Segregate data by age into **generations**.

- Arrange that the younger generations can be collected independently of the older ones.
- When space is needed, collect the youngest generation first.
- Only collect older generation(s) if space is still needed.
- Should make GC more efficient overall, since less total tracing is performed.
- Should shorten pause times (at least for young generation GCs).

Some variant of generational collection is almost universally used in serious implementations of heavily-allocating languages (LISP, functional languages, Smalltalk, ...)

Most generational systems are copying collectors, although mark and sweep variants are possible.

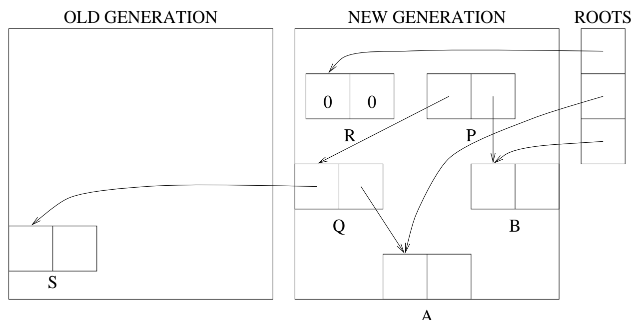
In generational copying collector, data in generation $n$ that are still live after a certain number of gc's (the **promotion threshold**) are copied into generation $n+1$ (possibly triggering a collection there).

Key problem: finding all the roots that point into generation $n$ without scanning higher generations.

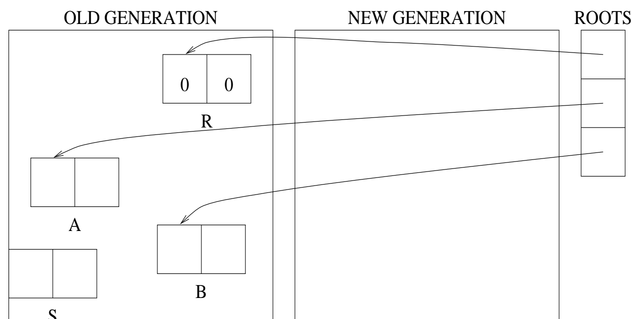


## Example

Assume 2 generations, promotion threshold = 1. Initial memory configuration after allocation of R:




Suppose a GC is now needed:

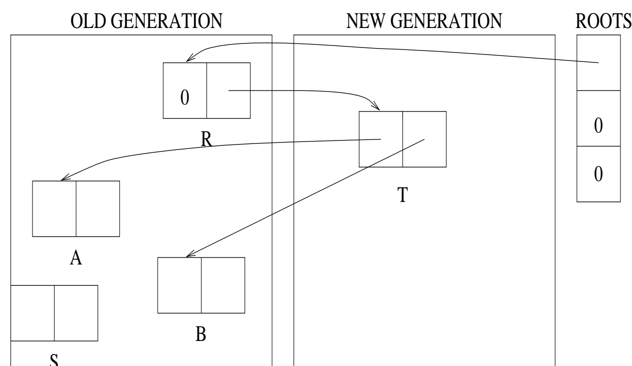



Note that S is now **tenured** (uncollected garbage).



## Example (continued)

Now we allocate a new cell T pointed to by R, fill T with pointers to A and B, and zero the root set pointers to A and B.




If a further GC is needed, we **must** follow the inter-generational pointer from R to T.



## Design issues

Tracking pointers from older generations to younger ones.

- This is primary added cost of generational system.
- Can only happen via **update** of a pointer in an older generation.
- Typically use a **write barrier** to catch such updates.
- Maintain **remembered set** of updated memory chunks ("cards"), where chunk size can range from single address to entire page.
- Hope there are not too many!
- Different tradeoffs in mutator overhead vs. scan time.

Promotion policy?

- Threshold = 1 gives simpler implementation, since no need to record object age, but promotes very young objects.

How many generations?

- Two-generation systems give simpler implementation, but multiple generations are useful if there is a spread of object lifetimes (especially if threshold = 1).

May want separate areas for large, pointer-free, or "immortal" objects.

## Incremental/Concurrent Collectors

Stop-and-copy collectors suffer from arbitrarily long **pauses** during collection, which is bad for interactive applications.

**Incremental** collectors attempt to solve this problem by doing the collection in small pieces, no one of which takes too long. Typically do some collection work at the time of each allocation. E.g., every time a cell is allocated, collector traces $k$ cells (for some $k > \frac{A}{M-A}$). Note: incremental $\neq$ **real-time**.

Because collector and mutator are interleaved at fine granularity, they must **synchronize** their activities. Useful to view collector and mutator as **concurrent** threads. In fact, can usually make incremental GC algorithms run on a multiprocessor, with one or more threads devoted to doing GC.

(Reference counting is essentially a concurrent GC approach.)

Key problem: mutator can change set of live cells "under the feet" of the collector. Example:

```
Mutator          Collector
q = new
r = new
p = q
                 trace p; mark q live
p = r
                 trace q... (ignore r)
```

**Essential** not to treat live data as garbage. **Desirable** not to treat garbage as live data.



## Tricolor Invariant

To describe essential invariant for concurrent marking, think of each heap cell as having one of three **colors**:

**Black** - node and its immediate children have been visited already.

**Grey** - node has been visited, but its children might not have been, so collector must revisit this node.

**White** - node has not been visited yet.

Initially, all nodes are white. A scan cycle terminates when all reachable nodes are black and there are no grey nodes; at this point, all remaining white nodes are garbage.

Example: In copying collector, old-space nodes are white, new-space nodes are grey until scanned, after which they become black.

Key idea: To maintain correctness of scan, mutatator must never write a pointer directly from a black node to a white node.

Many different ways to enforce this invariant.



## Baker's Algorithm (Incremental Copying Collector)

Idea: Never let the mutator "see" a white node (i.e., an old-space pointer). That way, mutator can never write such a pointer into a black node (i.e., a new-space node).

Implementation requires a **read barrier** that traps attempts to load from old-space cells, and moves them to new-space "on the fly."

Original mutator code:

```
x = p->c[1];
```

New mutator code after insertion of read barrier:

```
x = (forward(p))->c[1];
```

Also arranges to allocate new cells in new-space, beyond scan boundary (i.e., colored black); this is quite conservative, but safe and simple.

Read barrier makes this algorithm very expensive in the absence of hardware support. One possible variant: use memory management hardware to implement read barrier at page level.



## Dijkstra's Algorithm (Incremental Mark-Sweep Collector)

In a M&S collector, often use an explicit **mark stack** to record nodes that have been marked but whose children have not been:

```
struct cell *markstack[];
int msp = 0;
void gc () {
  for (int i = 0; i < ROOTS; i++) {
     roots[i]->mark = 1;
     markstack[msp++] = roots[i];
  }
  while (struct cell *cell = markstack[--msp])
    for (int child = 0; child <= 1; child++)
      if (!cell->c[child]->mark) {
         cell->c[child]->mark = 1;
         markstack[msp++] = cell->c[child];
      }
  sweep();
}
```

In this context, unmarked cells are white, marked cells that are on the markstack are grey, and other marked cells are black.

## Incremental Mark-Sweep (cont.)

The most direct way to prevent a black-to-white pointer from being written is to use a **write barrier** to detect such writes, and change the target object's color to grey "on the fly."

Original mutator code:

```
*p = q;
```

Mutator code with write barrier:

```
*p = q;
if (!q->mark) {
  q->mark = 1;
  markstack[msp++] = q;
}
```



## Garbage Collection in Java

Sun's original JVM uses a "mark and compact" collector

• Compromise between M&S and copying collectors.

• Live data cells are marked.

• Then heap is scanned and live data are slid down to a compact region at the bottom of the heap.

• Extra space costs for forwarding pointers; extra time costs for added traversals.

• Object pointers actually point to a **handle** which in turn points to the real object data. Handles are allocated in their own space, managed by M&S, and never moved. Object data records can be moved just by changing the handle's contents, without altering the object pointer.

Using handles supports conservative collection, which is handy because of bytecode subroutines and native code, but usually unnecessary.



## Garbage Collection and Native Code under Java

Interfacing to native code is problematic:

Java objects referenced by native code must not be collected by Java GC until native code is done with them.

Solutions:

• GC can implicitly register all arguments passed to native code (or returned to native code by callbacks into Java), and not collect them until native code finishes.
• References that need to live longer must be explicitly registered (and later unregistered) by native code programmer.

Java objects referenced directly by native code can't be moved.

Solutions:

• Use non-moving collector.
• Pass unmoveable, indirect object references to native code.
• Pin objects passed to native code.
• Make pinned copies of objects passed to native code.

Native code data objects pointed to by Java objects that get GC'ed should be freed.

Solution:

• Use **finalization** routines.