GARBAGE COLLECTION

Prof. Chris Jermaine
cmj4@cs.rice.edu

What Is Garbage Collection?

- A “garbage collected” PL implementation (like Java)...
  - — Provides automatic support for reclaiming memory you are done with
  - — The imp itself checks to see if there is any way to reach some allocated memory
  - — And if there is not, it deletes it
- Key thing to be aware of:
  - — It’s a bit strong to say a “language” itself is garbage collected (or not)
  - — Though that is what we (and everyone else) will say
  - — Why is the claim that “C is not garbage collected” a bit silly?
- What is alternative to GC?
  - — Ask the programmer to free memory when done with it. Always have:

```
SparseDoubleVector foo = new SparsDoubleVector (100, 1.0)
some code;
delete foo;
```

Short History of Garbage Collection

- Way back when, there was no dynamic memory allocation
  - — We're talking Fortran in the mid 1950's
  - — You declared all your vars up front, at compile time
  - — So no need for PL system to dynamically reclaim memory...
  - — In first implementations, compiler claimed space for each var
- GC was invented with Lisp in the late 1950s
  - — But remained a fringe idea
  - — Lisp was never tremendously popular
  - — Then, with advent of C/C++ in early 70's, self-memory management was "in"
- I'd say it only became mainstream with first Java in mid 90's
  - — Now, a very common idea

Pros And Cons of GC

- Pros? Cons?

Pros And Cons of GC

- Pros
  - — Don’t need to worry about adding “free” or “delete” statements to code
  - — Never have bugs related to dangling references/pointers (double frees, etc.)
- Often incorrectly stated as a “pro”
  - — Don’t need to worry about memory leaks
- Cons
  - — GC takes lots of system resources; GC programs are usually slower than non-GC
  - — Ever wonder why you got huge performance swings in the A3 test cases?
  - — GC programs tend to consume whatever RAM you give them

Basics of GC Design and Implementation

- How do they work?
- First, need to differentiate between different memory “types”
- One one hand, there’s the stuff that can’t be garbage collected
  - — There’s the memory used to allocate everything you can’t ever (or easily) reclaim
  - — Such as static member variables
  - — There’s the memory used to allocate local variables having one of the 8 + 1 types
  - — Will refer to this as “root memory”
- Then there’s the stuff that can
  - — This is almost everything else
  - — Allocated on the “heap” in Java

Basics of GC Design and Implementation

- A tracing GC maintains a set containing all refs (or pointers)
- A reference links two memory regions:
  - The memory region (usually an object) where the reference lives
  - The memory region (usually an object) that the reference points to
- So the set of all references gives you a “reachability” graph
  - Nodes are memory regions
  - Edges are references
- When an object can’t be reached from any “root memory”...
  - It can safely be deallocated

Example

```
class Foo {
  private Foo left, right;

  public Foo (Foo child, Foo otherChild) {
    left = child;
    right = otherChild;
  }
}

...
Foo fooOne = new Foo (null, null);
Foo fooTwo = new Foo (null, null);
Foo fooThree = new Foo (fooOne, fooTwo);
Foo fooFour = fooThree;
fooThree = null;
fooFour = null;
...
```

Basics of GC Design and Implementation

- What would a super-simple GC do?
- Incrementally maintain the reachability graph
  - — Whenever you have a new allocation, add a node to the graph
  - — Whenever an object/var goes out of scope, remove a node from the graph
  - — Whenever you assign a reference a new value, change the link out of the node
- Then when you run out of RAM
  - — Do a DFS on the graph to find everyone not reachable from some “root memory”
  - — Declare all of that unreachable memory as being available for reallocation

What'd Be the Problem With This?

- Too much waiting while the GC runs
- Program would seem to freeze for awhile
- Very annoying from the user's point of view

Modern GCs Tend to Use Some Variant of…

- ...The “tri-color” method. Just an extended, lazy DFS
- Each node in reachability graph has a color
  - — White: don’t know if reachable
  - — Gray: reachable, children not processed yet
  - — Black: reachable, children processed
- Root memory nodes are gray initially, all else white
- Then do the following forever:
  - — Pick gray node; if white child, color child gray
  - — If no white child, color black
- When gray is empty, white nodes can be killed
- Can be interleaved with program exec., so less long dead spells

Modern GCs Also “Generational”

- Might have “major” and “minor” sweeps
- On “minor” sweeps, only put recent stuff into white set
- How recent is measured by object “generation”

OK, So How Does This Affect You?

- Well, at one time or another, all of you have been annoyed at Java
  - — Why is it hanging, doing nothing? Now you know! Doing a DFS
- If performance is really key, you can't be overly naive re GC
  - — Java actually gives you some knobs and levers to push and pull in attempt to tune
  - — See http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

Some Other Issues Re GC In Java

- Like almost all GCs, Java uses “syntactic” collection
- The GC does not try to understand your program
  - — It is just naively following references
- So if you keep around a lot of references to stuff you’ll never need
  - — Java won’t/can’t understand that it is OK to reclaim the memory
- This means you need to think a bit about keeping live references
  - — Got a monster data structure you are done with?
  - — Put it in a program block where its reference will go out of scope
  - — Or set all references to it to “null”
  - — Also, be careful about putting references into long-lived containers
  - — All of this is why GC does **NOT** mean you can forget about memory management

Also, There's No Way to Force GC in Java

- Does exist a system call that "suggests" it
- But GC is free to ignore you
- Find yourself using that system call?
  - — Almost always a case of voodoo programming
- Java's GCs are quite good (for GCs, that is)
  - — If you've run out of RAM, it's not because the GC forgot to run
  - — Or because you forgot to suggest that it should
  - — It's because you are keeping references to too many objects that are too large
  - — So make sure you are not keeping around a gazillion worthless refs
  - — And may sure you are not inadvertently creating huge data structures
  - — Don't blame the GC!

Questions?