Programs as Data

Garbage collection techniques

Peter Sestoft

Monday 2013-10-21*

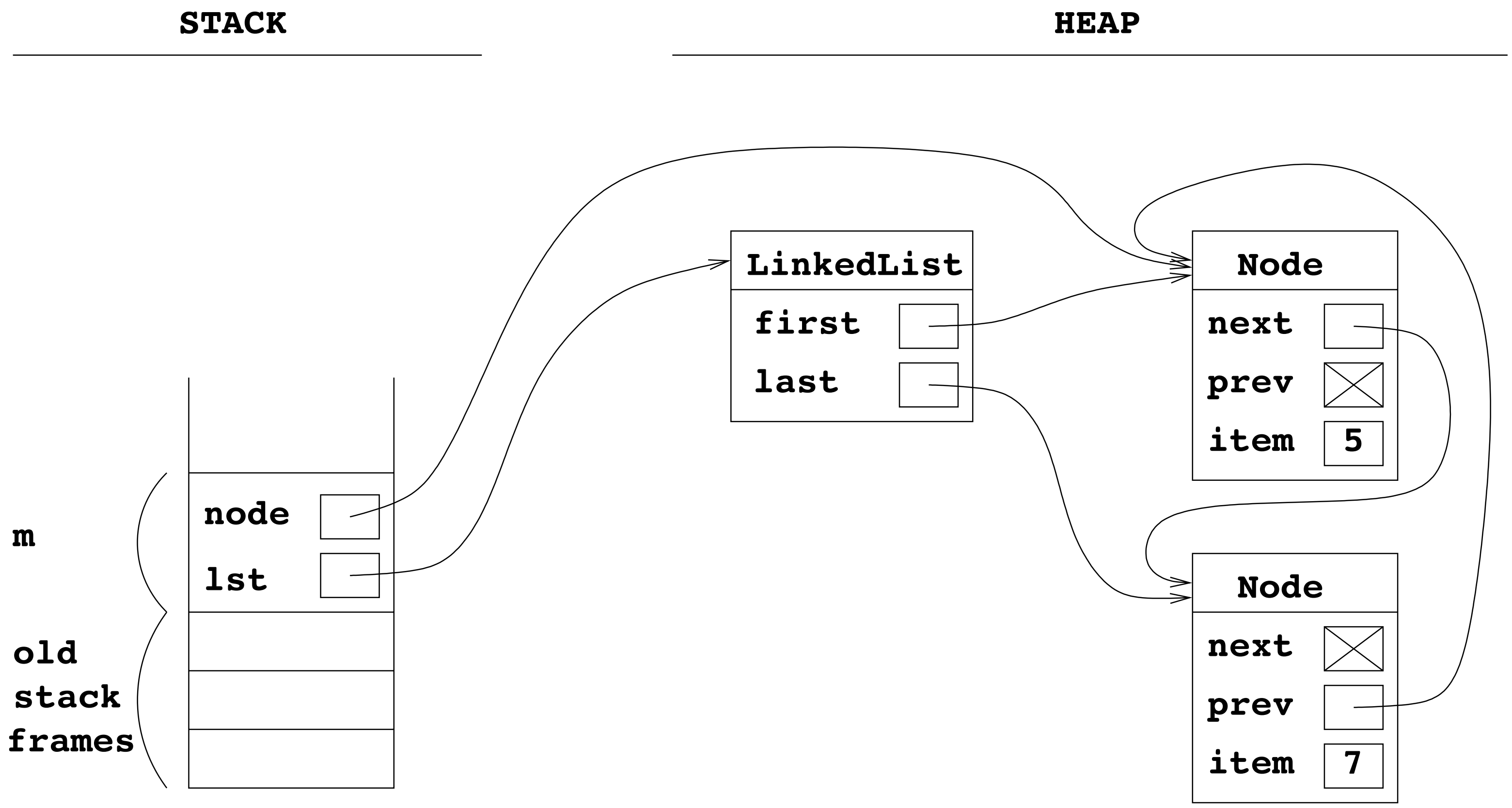
Note by Baris Aktemur:
These slides have been adapted from the originals available at http://www.itu.dk/courses/BPRD/E2013/.
I thank Peter Sestoft for making the PPT's available.

Garbage collection

- A: Reference counting
- B: Mark-sweep
- C: Two-space stop-and-copy, compacting
- D: Generational

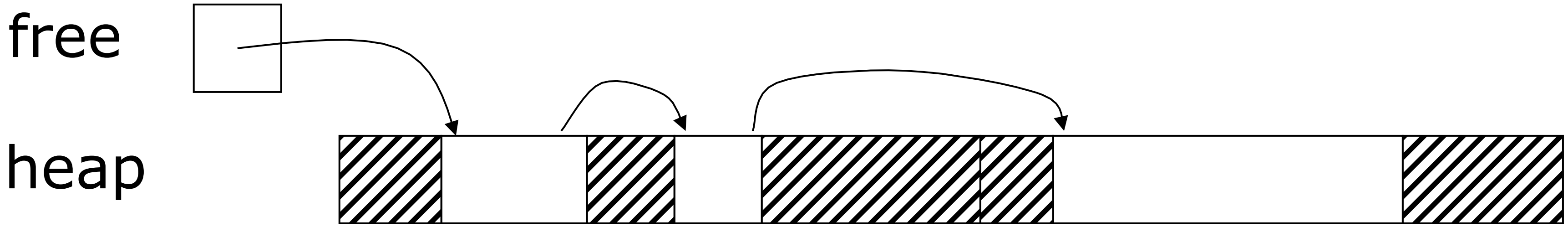
The heap as a graph

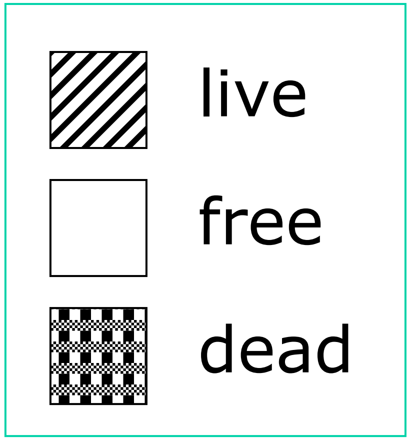
- The heap is a *graph*: node=object, edge=reference
- An object is *live* if reachable from *roots*
- Garbage collection *roots* = stack elements



The freelist

- A freelist is a linked list of free heap blocks:





- Allocation from freelist:
  - Search for a large enough free block
  - If none found, do garbage collection
  - Try the search again
  - If it fails, we are out of memory

A: Reference counting with freelist

- Each object knows the number of references to it
- Allocate objects from the freelist
- After assignment **`x=o;`** the runtime system
  - Increments the count of object o
  - Decrements the count of x's old reference (if any)
  - If that count becomes zero,
    - put that object on the freelist
    - recursively decrement count of all objects it points to
- Good
  - Simple to implement
- Bad
  - Reference count field takes space in every object
  - Reference count updates and checks take time
  - A cascade of decrements takes long time, gives long pause
  - Cannot deallocate cyclic structures

B: Mark-sweep with freelist

- Allocate objects from the freelist
- GC phase 1: mark phase
  - Assume all objects are white to begin with
  - Find all objects that are reachable from the stack, and color them black
- GC phase 2: sweep phase
  - Scan entire heap, put all white objects on the freelist, and color black objects white
- Good
  - Rather simple to implement
- Bad
  - Sweep must look at entire heap, also dead objects; inefficient when many small objects die young
  - Risk of *heap fragmentation*

C: Two-space stop and copy

- Divide heap into to-space and from-space
- Allocate objects in from-space
- When full, recursively move all reachable objects from from-space to the empty to-space
- Swap (empty) from-space with to-space
- Good
  - Need only to look at live objects
  - Good reference locality and cache behavior
  - Compacts the live objects: no fragmentation
- Bad
  - Uses twice as much memory as maximal live object size
  - Needs to update references when moving objects
  - Moving a large object (e.g. an array) is slow
  - Very slow (much copying) when heap is nearly full

D: Generational garbage collection

- Observation: Most objects die young
- Divide heap into *young* (nursery) and *old* generation
- Allocate in young generation
- When full, move live objects to old gen. (minor GC)
- When old gen. full, perform a (major) GC there
- Good
  - Recovers much garbage fast
- Bad
  - May suffer fragmentation of old generation (if mark-sweep)
  - Needs a write barrier test on field assignments:
    After assignment **`o.f=y`** where **`o`** in old and **`y`** in young,
    need to remember that **`y`** is live

Concurrent garbage collection

- In a multi-cpu machine, let one cpu run GC
- Complicated
  - Race conditions when allocating objects
  - Race conditions when moving objects
- Typically suspends threads at "GC safe" points
  - May considerably reduce concurrency (because one thread may take long to reach a safe point)

GC in mainstream virtual machines

- Sun/Oracle Hotspot JVM (client+server)
  - Three generations
  - When gen. 0 is full, move live objects to gen. 1
  - Gen. 1 uses two-space stop-and-copy GC; when objects get old they are moved to gen. 2
  - Gen. 2 uses mark-sweep with compaction
- IBM JVM (used in e.g. Websphere server)
  - Highly concurrent generational; see David Bacon's paper
- Microsoft .NET (desktop+server)
  - Three generation small-obj heap + large-obj heap
  - When gen. 0 is full, move to gen. 1
  - When gen. 1 is full, move to gen. 2
  - Gen. 2 uses mark-sweep with occasional compaction
- Mono .NET implementation
  - Boehm's conservative collector (still standard May 2012)
  - New two-generational (stop-and-copy plus M-S or S-&-C)

Other GC-related topics

- *Large object space*: Large arrays and other long-lived objects may be stored separately
- *Weak reference*: A reference that cannot itself keep an object live
- *Finalizer*: Code that will be executed when an object dies and gets collected (e.g. close file)
- *Resurrection*: A finalizer may make a dead object live again (yrk!)
- *Pinning*: When Java/C# exports a reference to C/C++ code, the object must be pinned; if GC moves it, the reference will be wrong

GC stress (StringConcatSpeed.java)

- What do these loops do? Which is better?

```
StringBuilder buf
  = new StringBuilder();
for (int i=0; i<n; i++)
  buf.append(ss[i]);
res = buf.toString();
```

```
String res = "";
for (int i=0; i<n; i++)
  res += ss[i];
```